Automatic Differentiation for Tensor Algebras

Technical Report

Sebastian Urban[*], Patrick van der Smagt[†]

1 November 2017

Contents

[*]Technical University Munich, surban@tum.de
[†]Volkswagen Group, smagt@brml.org

1 Introduction

[Kjolstad et al., 2017] proposed a tensor algebra compiler. It takes expressions that define a tensor element-wise, such as

$$f_{ij}(a, b, c, \boldsymbol{d}) = \exp\left[-\sum_{k=0}^{4} \left((a_{ik} + b_{jk})^2 \, c_{ii} + d_{i+k}^3\right)\right] ,$$

and generates the corresponding compute kernel code. The arguments can be either dense or sparse matrices.

For machine learning applications, especially deep learning, it is often necessary to compute the gradient of a loss function $l(a, b, c, \boldsymbol{d}) = l(f(a, b, c, \boldsymbol{d}))$ with respect to model parameters $a, b, c, \boldsymbol{d}$. Hence, if tensor compilers are to be applied in this field, it is necessary to derive expressions for the derivatives of element-wise defined tensors, i.e. expressions of the form $(\mathrm{d}a)_{ik} \triangleq \partial l / \partial a_{ik}$.

When the mapping between function indices and argument indices is not 1:1, special attention is required. For example, for the function $f_{ij}(x) = x_i^2$, the derivative of the loss w.r.t. x is $(\mathrm{d}x)_i \triangleq \partial l / \partial x_i = \sum_j (\mathrm{d}f)_{ij} \, 2 \, x_i$; the sum is necessary because index j does not appear in the indices of f. Another example is $f_i(x) = x_{ii}^2$, where x is a matrix; here we have $(\mathrm{d}x)_{ij} = \delta_{ij} \, (\mathrm{d}f)_i \, 2 \, x_{ii}$; the Kronecker delta is necessary because the derivative is zero for off-diagonal elements. Another indexing scheme is used by $f_{ij}(x) = \exp x_{i+j}$; here the correct derivative is $(\mathrm{d}x)_k = \sum_i (\mathrm{d}f)_{i,k-i} \, \exp x_k$, where the range of the sum must be chosen appropriately.

In this publication we present an algorithm that can handle any case in which the indices of an argument are an *arbitrary linear combination* of the indices of the function, thus all of the above examples can be handled. Sums (and their ranges) and Kronecker deltas are automatically inserted into the derivatives as necessary. Additionally, the indices are transformed, if required (as in the last example). The algorithm outputs a symbolic expression that can be subsequently fed into a tensor algebra compiler.

We first review the basic automatic differentiation algorithm (sections 2 and 3) and necessary algorithms for integer matrix inversion and for solving systems of linear inequalities (section 4). Then, in section 5, we show how to extend automatic differentiation to generate derivative expressions for element-wise defined tensor-valued functions. An example and numeric verification of our algorithm are presented in section 6.

An open source implementation of the described algorithm is provided at

`https://github.com/surban/TensorAlgDiff`.

Please cite this publication when using the provided code.

2 Symbolic Reverse Accumulation Automatic Differentiation

Every function f can be written as a composition of elementary functions such as addition, subtraction, multiplication, division, trigonometric function, the exponential function, the logarithm and so on. For now let us assume that the elementary functions take one or more *scalar* arguments; thus f will also be a function accepting scalar arguments. For example, the function $f(x_1, x_2) = \exp(x_1 + x_2)$ can be written as $f(x_1, x_2) = f_1(f_2(x_1, x_2))$ with parts $f_1(t) = \exp(t)$ and $f_2(t_1, t_2) = t_1 + t_2$. It is also possible that parts appear more than once in a function. As an example $f(x) = \sin(x^2) \cdot \cos(x^2)$ can be decomposed into $f(x) = f_1\big[f_2\big(f_4(x)\big), f_3\big(f_4(x)\big)\big]$ where the parts $f_1(s,t) = s \cdot t$, $f_2(t) = \sin(t)$, $f_3(t) = cos(t)$ are used once and $f_4(t) = t^2$ is used twice. A decomposition of a function into parts can be represented by a computational graph, that is a directed acyclic graph where each node represents a function part f_i and an edge between two node represents that the target node is used as the value to an argument of the source node. An exemplary computational graph for the function $f(x) = f_1\big[f_2\big(f_3\big(f_4(x), f_5(x)\big)\big)\big]$ is shown by the blue nodes in fig. 1.

Automatic differentiation is based on the well-known chain rule, which states that for a scalar function of the form $f(x) = g(h(x))$ the derivative can be written as

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial g}{\partial h} \frac{\partial h}{\partial x} \,.$$

Given a function f and its decomposition into parts f_i, the following algorithm uses reverse accumulation automatic differentiation to obtain a computational graph for the derivatives of f. Since $f(\boldsymbol{x}) = f_1(\dots)$ the derivative of f w.r.t. f_1 is

$$\frac{\partial f}{\partial f_1} = 1 \,. \tag{1}$$

Then iteratively do the following: Find a part f_i for which the derivative of all its consumers is available but $\partial f / \partial f_i$ is yet unknown. A part f_c is a consumer of part f_i, if f_i occurs as a direct argument to f_c in the function f. Thus, in the graphical representation of f part f_c is a consumer of f_i, if there exists an edge from f_c to f_i. Since the computational graph of a function is acyclic, there will always exist a part f_i for which this condition is fulfilled. Let $\mathrm{csmr}(f_i)$ be the set of consumers of part f_i. Following the chain rule, the derivative of f w.r.t. f_i is given by

$$\frac{\partial f}{\partial f_i} = \sum_{d \in \mathrm{csmr}(f_i)} \frac{\partial f}{\partial f_d} \frac{\partial f_d}{\partial f_i} \,. \tag{2}$$

Repeat this process until the derivatives w.r.t. all parts $\partial f / \partial f_i$ have been calculated. Once

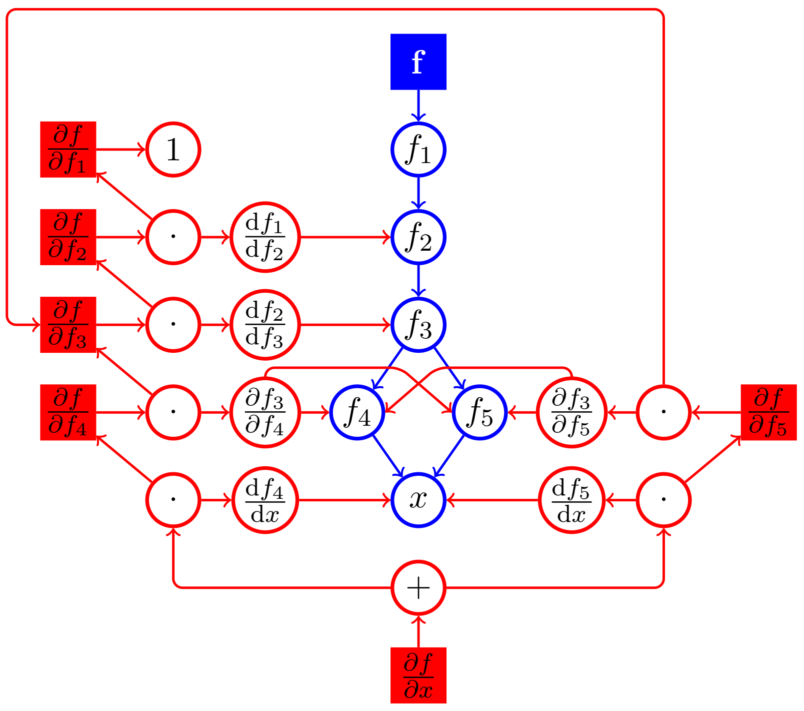

f
f_1
f_2
f_3
f_4
f_5
x
$\frac{\partial f}{\partial f_1}$
$\frac{\partial f}{\partial f_2}$
$\frac{\partial f}{\partial f_3}$
$\frac{\partial f}{\partial f_4}$
$\frac{\partial f}{\partial f_5}$
$\frac{\partial f}{\partial x}$
1
$\frac{\mathrm{d}f_1}{\mathrm{d}f_2}$
$\frac{\mathrm{d}f_2}{\mathrm{d}f_3}$
$\frac{\partial f_3}{\partial f_4}$
$\frac{\partial f_3}{\partial f_5}$
$\frac{\mathrm{d}f_4}{\mathrm{d}x}$
$\frac{\mathrm{d}f_5}{\mathrm{d}x}$
$\cdot$
$+$

Figure 1: The blue nodes show a computational graph for the function $f(x) = f_1\left[f_2\big(f_3\big(f_4(x), f_5(x)\big)\big)\right]$. Each node f_1, f_2, …, f_5 represents a part of the function and each edge represents an argument. By applying automatic differentiation as described in section 2 the computational graph for the derivatives (shown in red) is obtained.

completed, the derivatives of f w.r.t. its arguments x_j, $j \in \{1, \dots, n\}$, follow immediately,

$$\frac{\partial f}{\partial x_j} = \sum_{d \in \operatorname{csmr}(x_j)} \frac{\partial f}{\partial f_d} \frac{\partial f_d}{\partial x_j} \,. \tag{3}$$

Note, that this algorithm requires a single pass only to complete the derivatives of f w.r.t. to *all* of its parameters.

By performing this algorithm on the computational graph shown in fig. 1, the derivative represented by the red nodes and edges is obtained. The computation proceeds from top to bottom in a breadth-first order of traversation. In general the partial derivatives of the function parts can depend on all of its arguments, as it can be seen in the dependencies of the nodes for $\partial f_3 / \partial f_4$ and $\partial f_3 / \partial f_5$. Symbolic derivatives can be obtained from the resulting computational graph by starting from the node $\partial f / \partial x_i$ and following the dependencies until reaching the leafs of the graph. However, for numerical evaluation it is more efficient to insert numerical values for the parameters $\boldsymbol{x}$ into the graph and then evaluate it node by node. This ensures that intermediate values are only computed once and thus the possibility of an exponential blow up of the number of terms that can occur during classical symbolic differentiation is avoided. To evaluate the derivative $\partial f / \partial x$ numerically, the function $f(x)$ must be evaluated followed by the derivatives of all parts. This corresponds to the forward and backward passes of the backpropagation algorithm for neural networks.

An example of a computational graph and its derivative for the concrete function

$$f(x_1, x_2, x_3) = \sin\big[\sin\big(x_1 \cdot (x_2 + x_3) + \sinh(x_2 + x_3)\big)\big]$$

is shown in fig. 2.

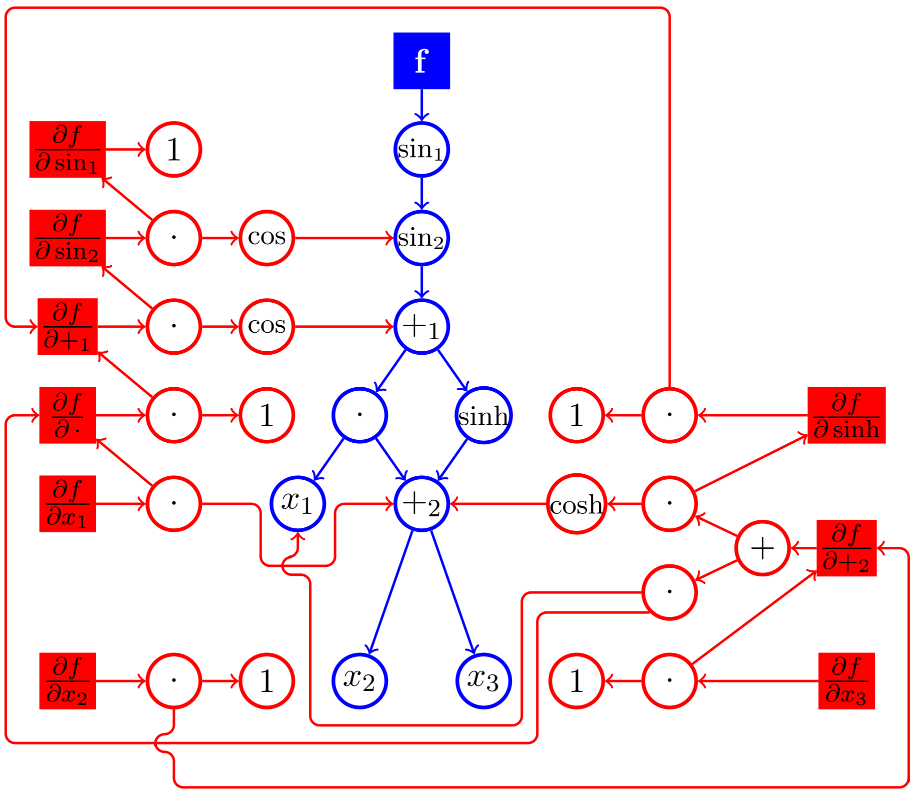

f
$\sin_1$
$\sin_2$
$+_1$
$\cdot$
sinh
x_1
$+_2$
x_2
x_3
$\frac{\partial f}{\partial \sin_1}$
$\frac{\partial f}{\partial \sin_2}$
$\frac{\partial f}{\partial +_1}$
$\frac{\partial f}{\partial \cdot}$
$\frac{\partial f}{\partial x_1}$
$\frac{\partial f}{\partial x_2}$
$\frac{\partial f}{\partial \sinh}$
$\frac{\partial f}{\partial +_2}$
$\frac{\partial f}{\partial x_3}$
1
cos
cos
cosh
$+$

Figure 2: A practical example for automatic symbolic differentiation. The computational graph for the function $f(x_1, x_2, x_3) = \sin\big[\sin\big(x_1 \cdot (x_2 + x_3) + \sinh(x_2 + x_3)\big)\big]$ is shown in blue. The computational graph for the derivatives obtained by automatic differentiation is shown in red. Note how intermediate values are reused automatically and the derivatives w.r.t. different x_i share most parts of the computational graph. Symbolic derivatives can be extracted from the graph or it can be evaluated numerically by substituting values for x_1, x_2 and x_3.

3 Handling Multidimensional Functions

So far we have shown automatic differentiation for scalar functions. However, in the context of artificial neural networks (ANNs) and Gaussian processs (GPs) we will mostly be dealing with functions that deal with tensor-valued functions. While any tensor-valued function can be written as a scalar function by splitting it into separate functions for each element of the tensor, doing so often has a significant penalty on computational efficiency. For example consider matrix multiplication. Calculating each element of $C = A \cdot B$ separately using $C_{ij} = \sum_k A_{ik} B_{kj}$ requires a total of $\mathcal{O}(n^3)$ operations where n is the size of the square matrices A and B. Contrary to that calculating all elements simultaneously can be done in $\mathcal{O}(n^{2.807})$ using the Strassen algorithm [Strassen, 1969] or even more efficiently in $\mathcal{O}(n^{2.375})$ using the Coppersmith-Winograd algorithm [Coppersmith and Winograd, 1987].[1] Thus we will show how to perform automatic differentiation on multidimensional functions now.

For functions working in two- or higher dimensional space, we use the vectorization operator vec to transform them into vector-valued functions. For a D-dimensional tensor $A \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_D}$ the vectorization operator is defined element-wise by

$$(\operatorname{vec} A)_{\sum_d s_d i_d} = A_{i_1, i_2, \dots, i_D}\,, \qquad i_d \in \{1, \dots, N_d\} \tag{4}$$

where the strides $\boldsymbol{s}$ are given by

$$s_d = \prod_{b=2}^{d} N_{b-1}\,.$$

As an example, for a matrix $A \in \mathbb{R}^{N \times M}$ this operator takes the columns of the matrix and stacks them on top of one another,

$$\operatorname{vec} A = \big(A_{11}, A_{21}, \dots, A_{N1}, A_{12}, A_{22}, \dots, A_{N2}, \dots, A_{1M}, A_{2M}, \dots, A_{NM}\big)^T\,.$$

Thus the derivatives of a tensor-valued function $F : \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_D} \to \mathbb{R}^{M_1 \times M_2 \times \cdots \times M_{D'}}$ can be dealt with by defining a helper function $\widehat{F} : \mathbb{R}^{N_1 N_2 \cdots N_D} \to \mathbb{R}^{M_1 M_2 \cdots M_{D'}}$ with $\widehat{F}(\operatorname{vec} X) = \operatorname{vec} F(X)$ and considering the derivatives of this vector-valued function $\widehat{F}$ instead.

It remains to show how to apply automatic differentiation to vector-valued functions. To do so, let us us first see how the chain rule works on vector-valued functions. Consider two functions, $\boldsymbol{g} : \mathbb{R}^K \to \mathbb{R}^N$ and $\boldsymbol{h} : \mathbb{R}^M \to \mathbb{R}^K$, and a composite function $\boldsymbol{f} : \mathbb{R}^M \to \mathbb{R}^N$ with $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{h}(\boldsymbol{x}))$. By expanding $\boldsymbol{g}(\boldsymbol{r})$ as $\boldsymbol{g}(r_1, r_2, \dots, r_K)$ and $\boldsymbol{h}(\boldsymbol{x})$

[1]While these algorithms are asymptotically faster than naive matrix multiplication, they also have a larger constant factor in their running time not captured by the big O notation. Therefore in practice they are only beneficial for matrices larger than a certain size.

as $\big(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \dots, h_K(\boldsymbol{x})\big)^T$ we can write

$$f_i(\boldsymbol{x}) = g_i\big(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \dots, h_K(\boldsymbol{x})\big)$$

and apply the chain rule on each argument of g_i, resulting in

$$\frac{\partial f_i}{\partial x_j} = \sum_{k=1}^{K} \frac{\partial g_i}{\partial h_k} \frac{\partial h_k}{\partial x_j} \,. \tag{5}$$

By introducing the Jacobian

$$\left(\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}}\right)_{ij} \triangleq \frac{\partial f_i}{\partial x_j}$$

we can rewrite (5) as a vectorized equation,

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} \,, \tag{6}$$

and thus obtain the chain rule for vector-valued functions. As we see, it is like the chain rule for scalars but with scalar multiplication replaced by matrix multiplication.

The algorithm for automatic differentiation for vector-valued functions is thus equal to scalar automatic differentiation described in section 2, but with eq. (1) replaced by

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{f_1}} = \mathbb{1} \tag{7}$$

and eq. (2) replaced by

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{f_i}} = \sum_{d \in \mathrm{csmr}(\boldsymbol{f_i})} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{f_d}} \frac{\partial \boldsymbol{f_d}}{\partial \boldsymbol{f_i}} \,. \tag{8}$$

For many common operations the size of the Jacobian $\partial \boldsymbol{f_d}/\partial \boldsymbol{f_i}$ may become very large. For example, the Jacobian of a matrix multiplication is of size n^4 for two matrices of size $n \times n$. However, since most elements are indeed zero, it is possible and vastly more efficient to directly compute the product $(\partial \boldsymbol{f}/\partial \boldsymbol{f_d})(\partial \boldsymbol{f_d}/\partial \boldsymbol{f_i})$ without explicitly evaluating the Jacobian. This is also the case for all elementary operations that work element-wise, such as addition, subtraction and the Hadamard product, which result in a diagonal Jacobian matrix. Consequently the explicit form (8) should only be used as a fall-back when such a shortcut computation is not available.

4 Systems of Integer Equalities and Inequalities

This section introduces methods to solve systems of integer equalities and inequalities. The algorithms presented here will be employed the compute the element-wise derivative expressions of tensor-valued functions.

4.1 Systems of Linear Integer Equations

Consider a system of linear equations

$$\begin{aligned} A_{11}\,x_1 + A_{12}\,x_2 + \cdots + A_{1m}\,x_m &= b_1 \\ A_{21}\,x_1 + A_{22}\,x_2 + \cdots + A_{2m}\,x_m &= b_2 \\ &\vdots\vdots \\ A_{n1}\,x_1 + A_{n2}\,x_2 + \cdots + A_{nm}\,x_m &= b_n\,, \end{aligned}$$

with integer coefficients $A \in \mathbb{Z}^{N \times M}$, integer variables $\boldsymbol{x} \in \mathbb{Z}^M$ and integer targets $\boldsymbol{b} \in \mathbb{Z}^N$. In matrix notation this system can be expressed much briefer as

$$A\,\boldsymbol{x} = \boldsymbol{b}\,. \tag{9}$$

To determine the set of solutions the matrix A must be transformed into Smith normal form, which is a diagonal matrix of the form

$$S = \operatorname{diag}(\alpha_1, \alpha_2, \dots, \alpha_R, 0, \dots, 0) \tag{10}$$

with the property that

$$\alpha_i \mid \alpha_{i+1}, \quad 1 \le i < r\,, \tag{11}$$

where $a \mid b$ should be read as "a *divides* b". Analogously $a \nmid b$ should be read as "a *does not divide* b". The number of non-zero entries R in the diagonal corresponds to the rank of A. It can be shown [Adkins and Weintraub, 1999] that for each non-zero matrix $A \in \mathbb{Z}^{N \times M}$ there exist invertible matrices $U \in \mathbb{Z}^{N \times N}$ and $V \in \mathbb{Z}^{M \times M}$ so that

$$S = U\,A\,V \tag{12}$$

where $S \in \mathbb{Z}^{N \times M}$ is the Smith normal form of A. Using the smith normal form, the equation system (9) can be rewritten as

$$S\,\boldsymbol{x}' = \boldsymbol{b}' \tag{13}$$

with

$$\boldsymbol{x} = V\,\boldsymbol{x}'\,, \tag{14}$$

$$\boldsymbol{b}' = U\,\boldsymbol{b}\,. \tag{15}$$

Since S is diagonal, the solutions can be read off from (13), as we describe in the following.

For the zero rows of S the corresponding entries of $\boldsymbol{b}'$ must also be zero, otherwise the equation system would be inconsistent and no solution exists. Thus for the system to be solvable we must have

$$C\,\boldsymbol{b} = \boldsymbol{0} \tag{16}$$

where $C \in \mathbb{Z}^{N-R\times N}$ with $C_{ij} = U_{R+i,j}$ is the sub-matrix consisting of the rows $R+1$ to N of U. It is called the cokernel of A.

For each non-zero entry α_i of S we must have

$$x'_i = \frac{b'_i}{\alpha_i} \tag{17}$$

and thus a solution exists only if b'_i is dividable by α_i. We can define a so-called pseudo-inverse $I : \mathbb{Q}^{M\times N}$ with

$$I \triangleq V\,S^\dagger\,U \tag{18}$$

where $S^\dagger \in \mathbb{Q}^{N\times M}$ is defined by

$$S^\dagger = \operatorname{diag}(1/\alpha_1, 1/\alpha_2, \dots, 1/\alpha_R, 0, \dots, 0)\,, \tag{19}$$

with the factors α_i given by (10). This pseudo-inverse has the property that $A\,I\,A = A$. Thus, for every $\boldsymbol{b}$ that is in the cokernel of A, we can obtain an $\boldsymbol{x}$ by setting $\boldsymbol{x} = I\,\boldsymbol{b}$ so that $A\,\boldsymbol{x} = \boldsymbol{b}$.

For the zero columns of S the corresponding entries of $\boldsymbol{x}'$ do not affect the value of $\boldsymbol{b}'$. Consequently, the columns of the matrix $K \in \mathbb{Z}^{M\times M-R}$, with $K_{ij} = V_{i,R+j}$, are a basis for the kernel (also called null-space) of A. This means that $M\,K = \boldsymbol{0}$ and thus every $\boldsymbol{b}$ that is in the cokernel of A we can write $\boldsymbol{b} = A(I\,\boldsymbol{b} + K\,\boldsymbol{z})$ where $\boldsymbol{z} \in \mathbb{Z}^{M-R}$ is a vector of arbitrary integers.

In summary, the equation system $A\,\boldsymbol{x} = \boldsymbol{b}$ has no integer solution for a particular $\boldsymbol{b}$, if $C\,\boldsymbol{b} \neq \boldsymbol{0}$ or $I\,\boldsymbol{b} \notin \mathbb{Z}^N$. Otherwise, if A has full rank, that is $R = N = M$, a unique integer solution exists, determined by $\boldsymbol{x} = I\,\boldsymbol{b}$. If A has non-full rank, infinitely many integer solutions exist and are given by $\boldsymbol{x} = I\,\boldsymbol{b} + K\,\boldsymbol{z}$ where $\boldsymbol{z} \in \mathbb{Z}^{M-R}$ is a vector of arbitrary integers.

4.1.1 Computation of the Smith Normal Form

An algorithm [Smith, 1860] that, given a matrix A, computes the Smith normal form S and two matrices U and V, such that $S = U\,A\,V$ is shown in algorithm 1. The algorithm transforms the matrix A into Smith normal form by a series of elementary row and column operations. Matrices U and V are initialized to be identity matrices and the same row and column operations are applied to them, so that in the end the relation $S = U\,A\,V$ holds. Since all operations are elementary, it follows that U and V are invertible as required. By following the description of the algorithm it is clear that the resulting matrix S will be diagonal and fulfill the property (11). To find the factors β, σ and τ of Bézout's identity in steps 10 and 19 the extended Euclidean algorithm [Knuth, 1997] is used, which is shown in algorithm 2.

What remains to show is that the described algorithm terminates. With each iteration of the loop in step 9 the absolute value of the element S_{aa} decreases, because it is replaced with the greatest common divisor (GCD) of itself and another element. Thus, this loop will terminate since, in worst case, $S_{aa} = +1$ or $S_{aa} = -1$ will divide all following rows and columns. The same argument holds, when the matrix must be rediagonalized due to the execution of step 36. It is easy to verify that the first diagonalization step executed thereafter will set $S_{aa} = \gcd(S_{aa}, S_{a+1,a+1})$ and thus the absolute value of S_{aa} decreases. Thus, in the worst case, the loop terminates as soon as $S_{11} = S_{22} = \cdots = S_{R-1,R-1} = 1$, which then divides S_{RR}.

4.2 Systems of Linear Inequalities

Consider a system of linear inequalities

$$\begin{aligned}
A_{11}\,x_1 + A_{12}\,x_2 + \cdots + A_{1M}\,x_M &\geq b_1 \\
A_{21}\,x_1 + A_{22}\,x_2 + \cdots + A_{2M}\,x_M &\geq b_2 \\
&\vdots \\
A_{N1}\,x_1 + A_{N2}\,x_2 + \cdots + A_{NM}\,x_M &\geq b_N\,,
\end{aligned} \tag{20}$$

with coefficients $A \in \mathbb{R}^{N \times M}$, variables $\boldsymbol{x} \in \mathbb{R}^M$ and biases $\boldsymbol{b} \in \mathbb{R}^N$. Note that this notation can also describe equalities by including the same line twice, where one occurrence is multiplied by -1 on both sides. In matrix notation this inequality system can be expressed much briefer as

$$A\,\boldsymbol{x} \geq \boldsymbol{b}\,. \tag{21}$$

Algorithm 1: Smith normal form of an integer matrix

Input: non-zero matrix $A \in \mathbb{Z}^{N\times M}$

Output: Smith normal form $S \in \mathbb{Z}^{N\times M}$, invertible matrices $U \in \mathbb{Z}^{N\times N}$, $V \in \mathbb{Z}^{M\times M}$, rank R

```
1  $U \longleftarrow \mathbb{1}_N$; $V \longleftarrow \mathbb{1}_M$          // initialize $U$ and $V$ with identity matrices
2  $S \longleftarrow A$                                      // initialize $S$ with $A$
3  $a \longleftarrow 1$                             // initialize active row and column
4  while $\exists i, j : i \geq a \wedge j \geq a \wedge S_{ij} \neq 0$ do                  // diagonalize $S$
       // Bring non-zero pivot element into position $S_{aa}$
5      Simultaneously $S_{\cdot a} \longleftarrow S_{\cdot j}$ and $S_{\cdot j} \longleftarrow S_{\cdot a}$
6      Simultaneously $V_{\cdot a} \longleftarrow V_{\cdot j}$ and $V_{\cdot j} \longleftarrow V_{\cdot a}$
7      Simultaneously $S_{a\cdot} \longleftarrow S_{i\cdot}$ and $S_{i\cdot} \longleftarrow S_{a\cdot}$
8      Simultaneously $U_{a\cdot} \longleftarrow U_{i\cdot}$ and $U_{i\cdot} \longleftarrow U_{a\cdot}$
9      while $S$ is changing do        // zero all elements below and right of $S_{aa}$
10         while $\exists i : i > a \wedge S_{aa} \nmid S_{ia}$ do          // ensure divisibility of rows
11             Find $\beta, \sigma, \tau$ so that $\beta = \gcd(S_{aa}, S_{ia}) = \sigma\, S_{aa} + \tau\, S_{ia}$.
12             $\gamma \longleftarrow \frac{S_{ia}}{\beta}$;   $\alpha \longleftarrow \frac{S_{aa}}{\beta}$
13             Simultaneously $S_{a\cdot} \longleftarrow \sigma\, S_{a\cdot} + \tau S_{i\cdot}$ and $S_{i\cdot} \longleftarrow -\gamma\, S_{a\cdot} + \alpha S_{i\cdot}$
14             Simultaneously $U_{a\cdot} \longleftarrow \sigma\, U_{a\cdot} + \tau U_{i\cdot}$ and $U_{i\cdot} \longleftarrow -\gamma\, U_{a\cdot} + \alpha U_{i\cdot}$
15         while $\exists i : i > a \wedge S_{ia} \neq 0$ do        // eliminate first element of rows
16             $f \longleftarrow \frac{S_{ia}}{S_{aa}}$
17             $S_{i\cdot} \longleftarrow S_{i\cdot} - f\, S_{a\cdot}$
18             $U_{i\cdot} \longleftarrow U_{i\cdot} - f\, U_{a\cdot}$
19         while $\exists j : j > a \wedge S_{aa} \nmid S_{aj}$ do        // ensure divisibility of columns
20             Find $\beta, \sigma, \tau$ so that $\beta = \gcd(S_{aa}, S_{aj}) = \sigma\, S_{aa} + \tau\, S_{aj}$.
21             $\gamma \longleftarrow \frac{S_{aj}}{\beta}$;   $\alpha \longleftarrow \frac{S_{aa}}{\beta}$
22             Simultaneously $S_{\cdot a} \longleftarrow \sigma\, S_{\cdot a} + \tau S_{\cdot j}$ and $S_{\cdot j} \longleftarrow -\gamma\, S_{\cdot a} + \alpha S_{\cdot j}$
23             Simultaneously $V_{\cdot a} \longleftarrow \sigma\, V_{\cdot a} + \tau V_{\cdot j}$ and $V_{\cdot j} \longleftarrow -\gamma\, V_{\cdot a} + \alpha V_{\cdot j}$
24         while $\exists j : j > a \wedge S_{aj} \neq 0$ do    // eliminate first element of columns
25             $f \longleftarrow \frac{S_{aj}}{S_{aa}}$
26             $S_{\cdot j} \longleftarrow S_{\cdot j} - f\, S_{\cdot a}$
27             $V_{\cdot j} \longleftarrow V_{\cdot j} - f\, V_{\cdot a}$
28     $a \longleftarrow a + 1$                                  // next diagonal element
29 $R \longleftarrow a - 1$                 // rank is number of non-zero diagonal elements
30 for $a \in \{1, \ldots, R\}$ do
31     if $S_{aa} < 0$ then                              // ensure positive diagonal
32         $S_{\cdot a} \longleftarrow -S_{\cdot a}$
33         $V_{\cdot a} \longleftarrow -V_{\cdot a}$
34     if $a \leq R - 1 \wedge S_{aa} \nmid S_{a+1,a+1}$ then        // ensure divisibility constraints
35         $S_{\cdot a} \longleftarrow S_{\cdot a} + S_{\cdot, a+1}$
36         $V_{\cdot a} \longleftarrow V_{\cdot a} + V_{\cdot, a+1}$
37         Go back to step 4.
```

Algorithm 2: Extended Euclidean algorithm

```
Input: positive numbers $a \in \mathbb{Z}^+$, $b \in \mathbb{Z}^+$
Output: factors $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $z \in \mathbb{Z}$ fulfilling Bézout's identity
        $z = \gcd(a, b) = a\,x + b\,y$
1  $r_0 \longleftarrow a$ ; $r_1 \longleftarrow b$
2  $s_0 \longleftarrow 1$ ; $s_1 \longleftarrow 0$
3  $t_0 \longleftarrow 0$ ; $t_1 \longleftarrow 1$
4  $i \longleftarrow 1$
5  while $r_i \neq 0$ do
6      $q \longleftarrow \frac{r_{i-1}}{r_i}$                                // integer division
7      $r_{i+1} \longleftarrow r_{i-1} - q\,r_i$
8      $s_{i+1} \longleftarrow s_{i-1} - q\,s_i$
9      $t_{i+1} \longleftarrow t_{i-1} - q\,t_i$
10     $i \longleftarrow i + 1$
11 $z \longleftarrow r_{i-1}$; $x \longleftarrow s_{i-1}$; $y \longleftarrow t_{i-1}$
```

The objective is to transform the inequality system into the form

$$
\begin{aligned}
\max(L^M \boldsymbol{b}) &\leq x_M &&\leq \min(H^M \boldsymbol{b}) && (22)\\
\max(L^{M-1}\boldsymbol{b} + \widehat{L}^{M-1}\boldsymbol{x}_M) &\leq x_{M-1} &&\leq \min(H^{M-1}\boldsymbol{b} + \widehat{H}^{M-1}\boldsymbol{x}_M) && (23)\\
\max(L^{M-2}\boldsymbol{b} + \widehat{L}^{M-2}\boldsymbol{x}_{M-1\,\dots\,M}) &\leq x_{M-2} &&\leq \min(H^{M-2}\boldsymbol{b} + \widehat{H}^{M-2}\boldsymbol{x}_{M-1\,\dots\,M}) && (24)\\
&\vdots &&\vdots\\
\max(L^2\boldsymbol{b} + \widehat{L}^2\boldsymbol{x}_{3\,\dots\,M}) &\leq x_2 &&\leq \min(H^2\boldsymbol{b} + \widehat{H}^2\boldsymbol{x}_{3\,\dots\,M}) && (25)\\
\max(L^1\boldsymbol{b} + \widehat{L}^1\boldsymbol{x}_{2\,\dots\,M}) &\leq x_1 &&\leq \min(H^1\boldsymbol{b} + \widehat{H}^1\boldsymbol{x}_{2\,\dots\,M})\,, && (26)
\end{aligned}
$$

so that the range of each element x_i can be determined sequentially. Here $\boldsymbol{x}_{i\,\dots\,j}$ should be read as the subvector of $\boldsymbol{x}$ starting at element i and including all elements up to (including) element j. Furthermore $\min \boldsymbol{z}$ and $\max \boldsymbol{z}$ mean the minimum or maximum element of a vector $\boldsymbol{z}$. The transformed system should be tight in the sense that given a subvector $\boldsymbol{x}_{M-s\,\dots\,M}$ which satisfies the first $s+1$ inequalities there must exist remaining elements $\boldsymbol{x}_{1\,\dots\,M-s-1}$ so that $\boldsymbol{x}$ satisfies all inequalities. This is equivalent to demanding that the transformed inequalities must not allow values for an element x_i so that the ranges of allowed values for other elements of $\boldsymbol{x}$ becomes empty. Obviously the matrices L^i, $\widehat{L}^i$, H^i and $\widehat{H}^i$ depend on A and must be determined.

Multiplying an inequality by a positive, non-zero factor will result in an equivalent system, where equivalent means that it has exactly the same set of solutions as the original system. Thus, by dividing each line i with $A_{i1} \neq 0$ by the factor $|A_{i1}|$ and rearranging, we can bring a system of the form (20) into the equivalent form

$$x_1 + \sum_{j=2}^{M} D_{hj} x_j \geq d_h \,, \quad h \in \{1, \dots, H\} \tag{27}$$

$$-x_1 + \sum_{j=2}^{M} E_{kj} x_j \geq e_k \,, \quad k \in \{1, \dots, K\} \tag{28}$$

$$\sum_{j=2}^{M} F_{lj} x_j \geq f_l \,, \quad l \in \{1, \dots, L\} \tag{29}$$

with $H + K + L = N$. It is clear that adding two inequalities will not reduce the set of solutions, i.e. if $\boldsymbol{x}$ is a solution to the inequalities $\boldsymbol{a}^T\boldsymbol{x} \geq \alpha$ and $\boldsymbol{b}^T\boldsymbol{x} \geq \beta$, then $\boldsymbol{x}$ is also a solution to the inequality $(\boldsymbol{a}+\boldsymbol{b})^T\boldsymbol{x} \geq \alpha + \beta$. Consequently by adding each inequality from (27) to each inequality from (28) and dropping the used inequalities we arrive at the reduced system with x_1 eliminated,

$$\sum_{j=2}^{M} (D_{hj} + E_{kj})\, x_j \geq d_h + e_k \,, \quad h \in \{1, \dots, H\},\, k \in \{1, \dots, K\} \,, \tag{30}$$

$$\sum_{j=2}^{M} F_{lj} x_j \geq f_l \,, \qquad l \in \{1, \dots, L\} \,, \tag{31}$$

which has at least the solutions $\boldsymbol{x}$ of the original system consisting of eqs. (27) to (29). Fourier and Motzkin [Dantzig and Eaves, 1973] observed that both system are indeed equivalent. To verify this, we have to show that for each solution $\boldsymbol{x}_{2\cdots M}$ of eqs. (30) and (31), there exists x_1 so that the combined $\boldsymbol{x}$ satisfies eqs. (27) to (29). From (27) and (28) we see that an x_1 satisfying the original system is given by

$$\min_k \left(\sum_{j=2}^{M} E_{kj} x_j - e_k \right) \geq x_1 \geq \max_h \left(-\sum_{j=2}^{M} D_{hj} x_j + d_h \right) \tag{32}$$

and rewriting (30) as

$$\sum_{j=2}^{M} E_{kj} x_j - e_k \geq -\sum_{j=2}^{M} D_{hj} x_j + d_h \,, \quad h \in \{1, \dots, H\},\, k \in \{1, \dots, K\} \tag{33}$$

shows that an x_1 with this property exists if the reduced system is satisfied.

By iteratively applying the reduction method just described, we can sequentially eliminate x_1, x_2 and so on up to x_M, as long as there exists at least one pair of inequalities with opposite signs for a specific x_i. If this is not the case, then the remaining $x_{i+1\,\ldots\,M}$ are not affected by these inequalities since a value for x_i can always be found after determining $x_{i+1\,\ldots\,M}$ because x_i is bounded from one side only; consequently when x_i occurs with positive or negative sign only, all inequalities containing x_i can be dropped to progress with the elimination. After x_M has been eliminated, what remains is a system of constant inequalities of the form

$$0 \geq f_l\,, \quad l \in \{1, \ldots, L\}\,. \tag{34}$$

If these inequalities contain a contradiction, i.e. if any f_l is positive, the original system of inequalities is inconsistent and the set of solutions for $\boldsymbol{x}$ is empty.

This elimination method gives rise to algorithm 3 which has been adapted from [Dantzig, 2016, Dantzig and Thapa, 2006a, Dantzig and Thapa, 2006b] to work on matrix A only and thus solving the system of inequalities for arbitrary $\boldsymbol{b}$. The algorithm produces matrices L^i, H^i and $\widehat{L}^i$, $\widehat{H}^i$ for $i \in \{1, \ldots, M\}$ that can be inserted into the inequalities (22) to (26) to subsequently obtain the ranges for each element of $\boldsymbol{x}$. It also outputs the feasibility matrix F, with the property that if $F\boldsymbol{b} \leq \mathbf{0}$, then there exist a solution for a particular $\boldsymbol{b}$.

Algorithm 3: Fourier-Motzkin elimination for a system of linear inequalities $A\,\boldsymbol{x} \geq \boldsymbol{b}$

```
Input: matrix $A \in \mathbb{R}^{N\times M}$
Output: matrices $L^i$, $H^i$ and $\widehat{L}^i$, $\widehat{H}^i$ for $i \in \{1,\dots,M\}$ for use in (22) to (26);
        feasibility matrix $F$
1  $B \longleftarrow \mathbb{1}_N$                                   // initialize $B$ with identity matrix
2  for $k \in \{1,\dots,M\}$ do                                      // loop over variables to eliminate
       // divide each row i by $|A_{ik}|$
3      for $i \in \{1,\dots N\}$ do
4          if $A_{ik} \neq 0$ then
5              $A_{i\cdot} \longleftarrow \frac{1}{|A_{ik}|}\, A_{i\cdot}$
6              $B_{i\cdot} \longleftarrow \frac{1}{|A_{ik}|}\, B_{i\cdot}$
       // extract solution matrices
7      $\zeta \longleftarrow \{i \in \mathbb{Z} \mid A_{ik} = 0\}$; $\phi \longleftarrow \{i \in \mathbb{Z} \mid A_{ik} = +1\}$; $\mu \longleftarrow \{i \in \mathbb{Z} \mid A_{ik} = -1\}$
8      $S \longleftarrow -$ columns $\{k+1,\dots,M\}$ of $A$
9      $L^k \longleftarrow$ rows $\phi$ of $B$; $H^k \longleftarrow -$ rows $\mu$ of $B$
10     $\widehat{L}^k \longleftarrow$ rows $\phi$ of $S$; $\widehat{H}^k \longleftarrow -$ rows $\mu$ of $S$
       // eliminate $x_k$
11     if $\phi = \emptyset \wedge \mu = \emptyset$ then
           // $x_k$ does not occur, nothing to eliminate
12     else if $\phi = \emptyset \vee \mu = \emptyset$ then
           // $x_k$ occurs with coefficient $+1$ or $-1$ only
13         $A \longleftarrow$ rows $\zeta$ of $A$; $B \longleftarrow$ rows $\zeta$ of $B$
14     else
           // $x_k$ occurs with coefficients $+1$ and $-1$
15         $A' \longleftarrow$ rows $\zeta$ of $A$; $B' \longleftarrow$ rows $\zeta$ of $B$
16         for $p \in \phi$ do
17             for $n \in \mu$ do
18                 $A' \longleftarrow A'$ with row $A_{p\cdot} + A_{n\cdot}$ appended
19                 $B' \longleftarrow B'$ with row $B_{p\cdot} + B_{n\cdot}$ appended
20         $A \longleftarrow A'$; $B \longleftarrow B'$
21 $F \longleftarrow B$                                              // inequalities with no variables left
```

5 Elementwise-defined Functions and their Derivatives

We introduce the situations that can occur when calculating the derivatives of elementwise-defined tensors using the following set of examples. Then we will describe a general method to derive expressions for the derivatives of elementwise-defined tensors, where the indices of the arguments are an arbitrary linear combination of the indices of the function output. Summations within these expressions are allowed.

Consider the vector-valued function $\boldsymbol{f}^1 : \mathbb{R}^N \to \mathbb{R}^N$, that is defined by specifying how each element of $\boldsymbol{f}^1(\boldsymbol{x})$ depends on the elements of its arguments $\boldsymbol{x}$. For example, a very simple example for such a function is

$$f_i^1(\boldsymbol{x}) = \sin x_i \,.$$

Here it is straightforward to see that its Jacobian is given by

$$\frac{\partial f_i^1}{\partial x_{i'}} = \delta_{i,i'} \, \cos x_{i'}$$

since element i of $\boldsymbol{f}^1$ only depends by element i of its arguments $\boldsymbol{x}$. Hence, the Kronecker delta was introduced in the above expression to make sure that $\partial f_i^1 / \partial x_{i'} = 0$ for $i \neq i'$.

Further assume that $\boldsymbol{f}$ is part of a scalar function l with $l(\boldsymbol{x}) = g(\boldsymbol{f}(\boldsymbol{x}))$ and the derivatives of l w.r.t. the elements of $\boldsymbol{x}$ are to be derived. The derivatives $\partial g / \partial f_i$ are supposed to be known. Let us introduce the notation

$$\mathrm{d}\bullet_\alpha = \frac{\partial l}{\partial \bullet_\alpha}$$

for the derivatives of l w.r.t. an element of a variable or function. In the context of deep learning this is the derivative we are usually interested in, since it provides the gradient of a loss function l and is thus used for minimization of the loss. The explicit computation of the Jacobians $\partial f_i / \partial x_j$ is usually not of interest since it wastes space.

We obtain for our function $\boldsymbol{f}^1(\boldsymbol{x})$,

$$\mathrm{d}f_{i'}^1 = \sum_i \frac{\partial g}{\partial f_i^1} \frac{\partial f_i^1}{\partial x_{i'}} = \sum_i \mathrm{d}g_i \, \delta_{i,i'} \, \cos x_i = \mathrm{d}g_{i'} \, \cos x_{i'} \,.$$

Let us now consider a slightly more complicated example given by the function $f^2 : \mathbb{R}^N \times \mathbb{R}^{N \times N} \to \mathbb{R}^{N \times N}$ of two arguments with the element-wise specification

$$f_{ij}^2(\boldsymbol{x}, y) = x_i \, y_{ij} \,.$$

The (extended) Jacobians w.r.t. x and y are given by

$$\frac{\partial f^2_{ij}}{\partial x_{i'}} = \delta_{i,i'}\, y_{i'j}\,, \qquad \frac{\partial f^2_{ij}}{\partial y_{i'j'}} = \delta_{i,i'}\, \delta_{j,j'}\, x_{i'}\,,$$

where the derivative w.r.t. x does not contain a Kronecker delta for index j, since it is not used to index variable x. Consequently application of the chain rule gives the following derivatives of l,

$$\mathrm{d}x_{i'} = \sum_j \mathrm{d}g_{i'j}\, y_{i'j}\,, \qquad \mathrm{d}y_{i'j'} = \mathrm{d}g_{i'j'}\, x_{i'}\,,$$

where the lack of index j on variable x has lead to a summation over this index. Another situation is demonstrated by the function $f^3 : \mathbb{R}^{N\times N} \to \mathbb{R}^N$ with

$$f^3_i(x) = x^3_{ii}\,.$$

The Jacobian,

$$\frac{\partial f^3_i}{\partial x_{i'j'}} = \delta_{i,i'}\, \delta_{i,j'}\, 3x^2_{i'j'}\,,$$

now contains two Kronecker deltas for the index i to express that $i = i' = j'$ must hold so that the derivative is non-zero. This leads to the derivative of l,

$$\mathrm{d}x_{i'j'} = \delta_{i',j'}\, \mathrm{d}g_{i'}\, 3x^2_{i'j'}\,,$$

which now contains a Kronecker delta itself, since it has not been canceled out by a corresponding summation. A good example for a function containing a summation over its arguments is the matrix dot product,

$$f^4_{ij}(x,y) = \sum_k x_{ik}\, y_{kj}\,,$$

which has the (extended) Jacobians

$$\frac{\partial f^4_{ij}}{\partial x_{i'k'}} = \delta_{i,i'} \sum_k \delta_{k,k'}\, y_{kj}\,, \qquad \frac{\partial f^4_{ij}}{\partial y_{k'j'}} = \delta_{j,j'} \sum_k \delta_{k,k'}\, x_{ik}\,.$$

Thus the derivatives of l evaluate to

$$\begin{aligned}
\mathrm{d}x_{i'k'} &= \sum_i \sum_j \mathrm{d}g_{ij}\,\delta_{i,i'}\,\sum_k \delta_{k,k'}\,y_{kj} = \sum_j \mathrm{d}g_{i'j}\,y_{k'j}\,, \\
\mathrm{d}y_{k'j'} &= \sum_i \sum_j \mathrm{d}g_{ij}\,\delta_{j,j'}\,\sum_k \delta_{k,k'}\,x_{ik} = \sum_i \mathrm{d}g_{ij'}\,x_{ik'}\,.
\end{aligned}$$

Note that the summation indices of the derivatives have changed over from k to j and i respectively. Finally consider the situation where the indices of the argument are given by a linear combination of the function indices, as demonstrated by $f^5 : \mathbb{R}^{N\times M} \to \mathbb{R}^{NM}$ with

$$f^5_{ij}(\boldsymbol{x}) = \exp x_{Mi+j}\,.$$

Its Jacobian is straightforward to express,

$$\frac{\partial f^5_{ij}}{\partial x_{i'}} = \delta_{Mi+j,i'}\,\exp x_{Mi+j}\,,$$

however to efficiently express the derivative of l,

$$\mathrm{d}x_{i'} = \sum_i \sum_j \mathrm{d}g_{ij}\,\delta_{Mi+j,i'}\,\exp x_{Mi+j}\,,$$

the occurring Kronecker delta should be combined with one of the sums, because one of them is redundant. To do so it is necessary to solve the equation $Mi + j = i'$ for j, which is trivial in this example. The solution is given by $j = i' - Mi$ and after substitution this results in

$$\mathrm{d}x_{i'} = \sum_i \mathrm{d}g_{i,i'-Mi}\,\exp x_{i'}\,.$$

Note that the sum range must be chosen appropriately, which is not shown here. We have seen that, depending on the constellation of indices of the arguments of a elementwise-defined function, the derivative will either introduce additional summations, drop existing summations, introduce Kronecker deltas or even require substitution of the solution of a linear equation system into the indices or a combination of these things.

5.1 Computing element-wise derivative expressions

We first describe the method without accounting for summations inside the function and reintroduce them later. Generally the problem of computing expressions for elementwise derivatives can be stated as follows. Let $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_{D_f})$ be a multi-index and let the tensor-valued function $f : \mathbb{R}^{N_1^1 \times \cdots \times N_{D_1}^1} \times \cdots \times \mathbb{R}^{N_1^P \times \cdots \times N_{D_P}^P} \to \mathbb{R}^{N_1^f \times \cdots \times N_{D_f}^f}$ taking P tensor arguments called $x^1, x^2, \dots, x^P$ be specified element-wise,

$$f_{\boldsymbol{\alpha}}(x^1, x^2, \dots, x^P) = \overline{f}(x^1_{A^1\boldsymbol{\alpha}}, x^2_{A^2\boldsymbol{\alpha}}, \dots, x^P_{A^P\boldsymbol{\alpha}}) \,, \tag{35}$$

where each matrix $A^p : \mathbb{Z}^{D_f} \to \mathbb{Z}^{D_p}$ maps from the indices of f to the indices of its argument x^p. Such a linear transform covers all the cases shown in the introductory examples. If the same argument x^p should appear multiple times with different indices, we shall treat it as different arguments (by renaming the different occurrences) and sum over the resulting expressions for the derivatives after they have been obtained. Note that $\overline{f} : \mathbb{R} \times \cdots \times \mathbb{R} \to \mathbb{R}$ is a *scalar* function. Furthermore let $g : \mathbb{R}^{N_1^f \times \cdots \times N_{D_f}^f} \to \mathbb{R}$ be a scalar-valued function and let $l = g \circ f$. Let $\mathrm{d}f \in \mathbb{R}^{N_1^f \times \cdots \times N_{D_f}^f}$ be the tensor of derivatives of l w.r.t. the elements of f, thus by above definition

$$\mathrm{d}f_{\boldsymbol{\alpha}} = \frac{\partial l}{\partial f_{\boldsymbol{\alpha}}} = \frac{\partial g}{\partial f_{\boldsymbol{\alpha}}} \,. \tag{36}$$

The objective is to obtain expressions that specify the derivatives of l w.r.t. the elements of each x^p element-wise, i.e.

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \frac{\partial l}{\partial x^p_{\boldsymbol{\beta}^p}} \tag{37}$$

where $\boldsymbol{\beta}^p = (\beta_1^p, \beta_2^p, \dots, \beta_{D_p}^p)$ is a multi-index enumerating the elements of x^p.

Applying the chain rule to (37) gives

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \frac{\partial l}{\partial x^p_{\boldsymbol{\beta}^p}} = \sum_{\substack{\mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f \\ A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p}} \frac{\partial l}{\partial f_{\boldsymbol{\alpha}}} \frac{\partial f_{\boldsymbol{\alpha}}}{\partial x^p_{\boldsymbol{\beta}^p}} = \sum_{\boldsymbol{\alpha} \in \Gamma(\boldsymbol{\beta}^p)} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \overline{f}}{\partial x^p_{A^p\boldsymbol{\alpha}}} \tag{38}$$

and since $\overline{f}$ is a scalar function, computing the scalar derivative $\partial \overline{f} / \partial x^p_{A^p\boldsymbol{\alpha}}$ is straightforward using the strategy described in section 2. Thus the main challenge is to efficiently evaluate the summation over the set

$$\Gamma(\boldsymbol{\beta}^p) = \{\boldsymbol{\alpha} \in \mathbb{Z}^{D_f} \mid \mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f \ \wedge \ A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p\} \,, \tag{39}$$

i.e. find all *integer* vectors $\boldsymbol{\alpha}$ that fulfill the relation $A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p$ and lie within the range $\mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f$ determined by the shape of f.

An elementary approach, as demonstrated in the introductory examples, is to rewrite eq. (38) as

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \sum_{\alpha_1=1}^{N^1} \cdots \sum_{\alpha_{D_f}=1}^{N^{D_f}} \delta_{A^p\boldsymbol{\alpha}-\boldsymbol{\beta}^p}\, \mathrm{d}f_{\boldsymbol{\alpha}}\, \frac{\partial \overline{f}}{\partial x^p_{A^p\boldsymbol{\alpha}}} \tag{40}$$

where the single-argument Kronecker delta is given by $\delta_{\boldsymbol{t}} = 0$ for $\boldsymbol{t} \neq \boldsymbol{0}$ and $\delta_{\boldsymbol{0}} = 1$. Thus for each index $\boldsymbol{\alpha}$ of f we test explicitly if it contributes to the derivative of index $\boldsymbol{\beta}^p$ of argument x^p and if so, we include that element in the summation. By evaluating (40) for all $\boldsymbol{\beta}^p$ in parallel the cost of iterating over $\boldsymbol{\alpha}$ can be amortized over all elements of $\mathrm{d}x^p$. However, if multiple threads are used to perform this iteration, as it is required to gain acceptable performance on modern GPUs, locking becomes necessary to serialize writes to the same element of $\mathrm{d}x^p$. If A^p has low rank, write collisions on $\mathrm{d}x^p_{A^p\boldsymbol{\alpha}}$ become likely, leading to serialization and thus considerable performance loss.[2] Another drawback of this approach is that even if only a subset of elements of the derivative $\mathrm{d}x^p$ are required, the summation must always be performed over the whole range of $\boldsymbol{\alpha}$. Furthermore, while not being of great importance for minimization of loss functions in machine learning, it is regrettable that no symbolic expression for $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$ is obtained using this method.

For these reasons it is advantageous to find a form of the set eq. (39) that directly enumerates all $\boldsymbol{\alpha}$ belonging to a particular $\boldsymbol{\beta}^p$. This requires solving $A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p$ for $\boldsymbol{\alpha}$. In general, a set of linear equations with integer coefficients over integer variables, has either none, one or infinitely many solutions. The set of solutions can be fully described using the pseudo-inverse, cokernel and kernel. Thus, let I be the pseudo-inverse, C the cokernel and K the kernel of the integer matrix A as defined in section 4.1. Using these matrices we can rewrite (39) as

$$\Gamma(\boldsymbol{\beta}^p) = \{I\boldsymbol{\beta}^p + K\boldsymbol{z} \mid C\boldsymbol{\beta}^p = \boldsymbol{0} \wedge I\boldsymbol{\beta}^p \in \mathbb{Z}^{D_f} \wedge \boldsymbol{z} \in \mathbb{Z}^{\kappa} \wedge \boldsymbol{1} \leq I\boldsymbol{\beta}^p + K\boldsymbol{z} \leq \boldsymbol{N}^f\}\,, \tag{41}$$

where κ is the dimensionality of the kernel of A. The conditions $C\boldsymbol{\beta}^p = \boldsymbol{0}$ and $I\boldsymbol{\beta}^p \in \mathbb{Z}^{D_f}$ determine whether the set is empty or not for a particular $\boldsymbol{\beta}^p$ and since they are independent of $\boldsymbol{z}$, they only need to be checked once for each $\boldsymbol{\beta}^p$. Thus if these conditions do not hold, we can immediately conclude that $\mathrm{d}x^p_{\boldsymbol{\beta}^p} = 0$. Otherwise, in order to further simplify the set specification, we need to find the elements of the set

$$\Sigma(\boldsymbol{\beta}^p) = \{\boldsymbol{z} \in \mathbb{Z}^{\kappa} \mid \boldsymbol{1} \leq I\boldsymbol{\beta}^p + K\boldsymbol{z} \leq \boldsymbol{N}^f\} \tag{42}$$

containing all $\boldsymbol{z}$ that generate values for $\boldsymbol{\alpha}$ within its valid range. Since $\boldsymbol{\alpha}(\boldsymbol{z}) = I\boldsymbol{\beta}^p + K\boldsymbol{z}$

[2]The CUDA programming guide [Nvidia, 2017] is vague about the performance penalties associated with atomic addition to the same memory access from within multiple threads. Nonetheless, experiments [Farzad, 2013, yidiyidawu, 2012] show that performance can be degraded by up to a factor of 32 due to locking and resulting serialization.

is an affine transformation, the set $\Sigma(\boldsymbol{\beta}^p)$ must be convex. By rewriting the system of inequalities defining the set $\Sigma(\boldsymbol{\beta}^p)$ as

$$K\,\boldsymbol{z} \geq \mathbf{1} - I\boldsymbol{\beta}^p \tag{43}$$
$$-K\,\boldsymbol{z} \geq -\boldsymbol{N}^f + I\boldsymbol{\beta}^p \tag{44}$$

we can apply the Fourier-Motzkin algorithm described in section 4.2 to obtain the boundaries of the convex set in closed form. The Fourier-Motzkin algorithm produces matrix L^i, H^i and $\widehat{L}^i$, $\widehat{H}^i$ so that (42) can be written as

$$\begin{aligned}\Sigma(\boldsymbol{\beta}^p) = \{\boldsymbol{z} \in \mathbb{Z}^\kappa\ | &\lceil \max(L^\kappa \boldsymbol{b})\rceil \leq z_\kappa \leq \lfloor \min(H^\kappa \boldsymbol{b})\rfloor \wedge \\ &\lceil \max(L^{\kappa-1}\boldsymbol{b} + \widehat{L}^{\kappa-1}\boldsymbol{z}_\kappa)\rceil \leq z_{\kappa-1} \leq \lfloor \min(H^{\kappa-1}\boldsymbol{b} + \widehat{H}^{\kappa-1}\boldsymbol{z}_\kappa)\rfloor \wedge \\ &\cdots \wedge \\ &\lceil \max(L^1\boldsymbol{b} + \widehat{L}^1\boldsymbol{z}_{2\ldots\kappa})\rceil \leq z_1 \leq \lfloor \min(H^1\boldsymbol{b} + \widehat{H}^1\boldsymbol{z}_{2\ldots\kappa})\rfloor\}\end{aligned} \tag{45}$$

where

$$\boldsymbol{b}(\boldsymbol{\beta}^p) \triangleq \begin{bmatrix} \mathbf{1} - I\boldsymbol{\beta}^p \\ -\boldsymbol{N}^f + I\boldsymbol{\beta}^p \end{bmatrix}$$

and $\lfloor\bullet\rfloor$ and $\lceil\bullet\rceil$ are the floor and ceiling respectively. Since the Fourier-Motzkin algorithm executes independently of the value of $\boldsymbol{b}$, the computationally intensive procedure of computing the matrices L^i, H^i and $\widehat{L}^i$, $\widehat{H}^i$ is only done once for each kernel matrix K. Afterwards, computing the boundaries for a particular index $\boldsymbol{\beta}^p$ requires only four matrix multiplications per dimension and the determination of the minimum and maximum value of a vector.

An example for a one-dimensional kernel, i.e. line, is shown in fig. 3. In this case (45) consists of only one condition for z_1 and describes the part of the line that is inside the range specified by $\mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f$. Another example, this time for a two-dimensional kernel, i.e. plane, is shown in fig. 4. Due to the choice of the kernel basis, the range specified by $\mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f$ becomes a parallelogram in the domain of the kernel and thus the resulting ranges for z_1 and z_2 are dependent on each other.

Since we have the convex set

$$\begin{aligned}\Gamma(\boldsymbol{\beta}^p) &= \{I\boldsymbol{\beta}^p + K\boldsymbol{z}\ |\ C\boldsymbol{\beta}^p = \mathbf{0}\ \wedge\ I\boldsymbol{\beta}^p \in \mathbb{Z}^{D_f}\ \wedge\ \boldsymbol{z} \in \Sigma(\boldsymbol{\beta}^p)\} \\ &= \{I\boldsymbol{\beta}^p + K\boldsymbol{z}\ |\ C\boldsymbol{\beta}^p = \mathbf{0}\ \wedge\ \boldsymbol{z} \in \Sigma(\boldsymbol{\beta}^p)\}\,,\end{aligned} \tag{46}$$

where we were able to drop the integer condition on $I\boldsymbol{\beta}^p$, because it is redundant to $\Sigma(\boldsymbol{\beta}^p)$ being not empty, we can now expand the sum in eq. (38) and thus write down an explicit

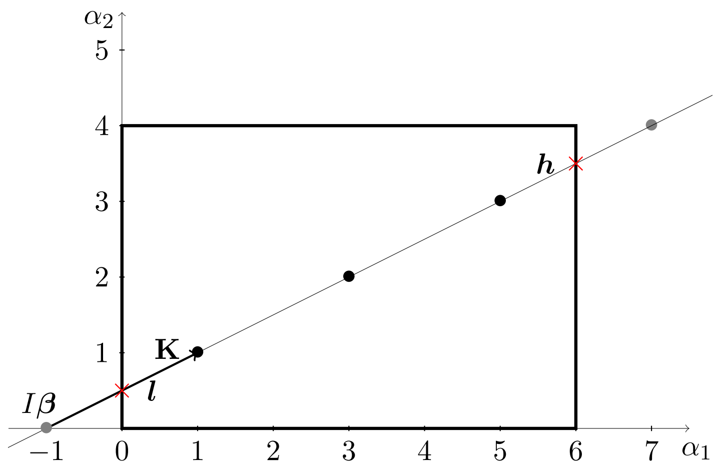
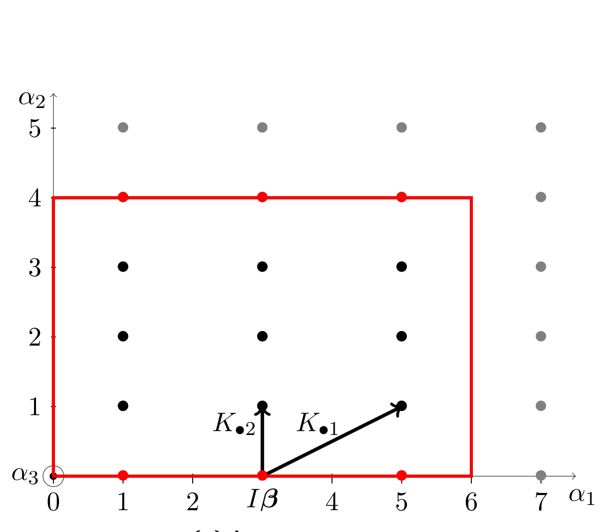


Figure 3: A one-dimensional parameter index $\boldsymbol{\beta}$ driven by a two-dimensional function index $\boldsymbol{\alpha}$ shown in $\boldsymbol{\alpha}$-space. The one-dimensional index of x is given by $\boldsymbol{\beta} = A\boldsymbol{\alpha}$ with $A = \begin{pmatrix}1 & -2\end{pmatrix}$. This yields $\boldsymbol{\alpha} = I\boldsymbol{\beta} + K\boldsymbol{z}$ with the pseudo-inverse $I^T = \begin{pmatrix}1 & 0\end{pmatrix}$ and one-dimensional kernel $K^T = \begin{pmatrix}2 & 1\end{pmatrix}$. For $\boldsymbol{\beta} = \begin{pmatrix}-1\end{pmatrix}$ the set of possible values for $\boldsymbol{\alpha}$ lies on the marked line with direction vector given by the kernel K. This set is limited by the requirement that $\boldsymbol{\alpha}$ must be integer, thus only the marked points on the line are valid values for $\boldsymbol{\alpha}$. Furthermore the constraint (42) imposed by the range of $\boldsymbol{\alpha}$ requires valid values to lie between the points marked $\boldsymbol{l}$ and $\boldsymbol{h}$. Thus values for z as allowed by (45) are $\Sigma(\begin{pmatrix}-1\end{pmatrix}) = \{z \in \mathbb{Z} \mid 1 \leq z \leq 3\}$, corresponding to the three points on the line inside the rectangle.

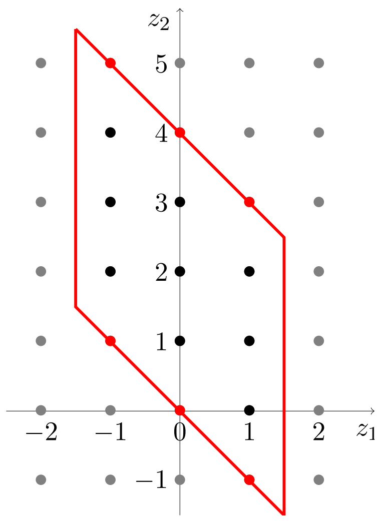
expression for $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$. This gives

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \delta_{C\boldsymbol{\beta}^p} \sum_{z_\kappa = \lceil \max(L^\kappa \boldsymbol{b}) \rceil}^{\lfloor \min(H^\kappa \boldsymbol{b}) \rfloor} \sum_{z_{\kappa-1} = \lceil \max(L^{\kappa-1}\boldsymbol{b} + \widehat{L}^{\kappa-1}\boldsymbol{z}_\kappa) \rceil}^{\lfloor \min(H^{\kappa-1}\boldsymbol{b} + \widehat{H}^{\kappa-1}\boldsymbol{z}_\kappa) \rfloor} \cdots \sum_{z_1 = \lceil \max(L^1\boldsymbol{b} + \widehat{L}^1 \boldsymbol{z}_{2\,\ldots\,\kappa}) \rceil}^{\lfloor \min(H^1\boldsymbol{b} + \widehat{H}^1 \boldsymbol{z}_{2\,\ldots\,\kappa}) \rfloor} \mathrm{d}f_{I\boldsymbol{\beta}^p + K\boldsymbol{z}} \left. \frac{\partial \overline{f}}{\partial x^p_{A^p\boldsymbol{\alpha}}} \right|_{\boldsymbol{\alpha} = I\boldsymbol{\beta}^p + K\boldsymbol{z}}, \tag{47}$$

where no Kronecker delta occurs within the sums and thus all iterations are utilized. The evaluation of the sums can be parallelized without difficulty and no synchronization is necessary for writes to $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$ since in this form one thread can be used per element of $\mathrm{d}x^p$.



(a) in $\boldsymbol{\alpha}$-space



(b) in $\boldsymbol{z}$-space

Figure 4: A one-dimensional parameter index $\boldsymbol{\beta}$ driven by a three-dimensional function index $\boldsymbol{\alpha}$. (a) This shows $\boldsymbol{\alpha}$-space as a cut through the α_1-α_2 plane, i.e. the α_3-axis is perpendicular to this drawing. The one-dimensional index of x is given by $\boldsymbol{\beta} = A\boldsymbol{\alpha}$ with $A = \begin{pmatrix} 1 & -2 & -2 \end{pmatrix}$. This yields $\boldsymbol{\alpha} = I\boldsymbol{\beta} + K\boldsymbol{z}$ with the pseudo-inverse $I = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$. A possible choice for the two-dimensional kernel is $K = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 2 & 1 \end{pmatrix}$. For $\boldsymbol{\beta} = \begin{pmatrix} 3 \end{pmatrix}$ the set of possible values for $\boldsymbol{\alpha}$ is given by the sum of $I\boldsymbol{\beta}$ and integer linear combinations of the columns of the kernel matrix K. The constraint (42) imposed by the range of $\boldsymbol{\alpha}$ requires valid values to lie inside the red rectangle. (b) By mapping this rectangle into the domain of the kernel, i.e. $\boldsymbol{z}$-space, we obtain a parallelogram. Thus values for $\boldsymbol{z}$ as allowed by (45) are the integer points that lie within this parallelogram, i.e. $\Sigma(\begin{pmatrix} 3 \end{pmatrix}) = \{z \in \mathbb{Z} \mid -1 \leq z_2 \leq 5 \land \max(-1, -z_2) \leq z_1 \leq \min(1, 4 - z_2)\}$ corresponding to the 15 points inside the rectangle in $\boldsymbol{\alpha}$-space. This causes the range of z_1 to become dependent on the value of z_2.

5.2 Handling expressions containing sums

As mentioned earlier we also want to handle expressions that contain summations over one or more indices. For this purpose consider a function containing a summation depending on arguments $y^1, \dots, y^{P'}$. It can be written in the form

$$f_{\boldsymbol{\alpha}}(y^1, \dots, y^{P'}, x^1, \dots, x^P) = \overline{f}\left(\overline{s}(y^1, \dots, y^{P'}), x^1_{A^1\boldsymbol{\alpha}}, \dots, x^P_{A^P\boldsymbol{\alpha}}\right) \tag{48}$$

with

$$\overline{s}(y^1, \dots, y^{P'}) = \sum_{k \in \Psi} s(y^1_{\widehat{A}^1\widehat{\boldsymbol{\alpha}}}, \dots, y^{P'}_{\widehat{A}^{P'}\widehat{\boldsymbol{\alpha}}}) \tag{49}$$

where $s : \mathbb{R} \times \dots \times \mathbb{R} \to \mathbb{R}$ and $\widehat{\boldsymbol{\alpha}} \triangleq \begin{bmatrix} \boldsymbol{\alpha} & k \end{bmatrix}$ and $\widehat{A}^{p'} : \mathbb{Z}^{D_f+1} \to \mathbb{Z}^{D_{p'}}$ and $\Psi \subset \mathbb{Z}$ is a convex integer set. Using the chain rule to calculate the derivative of l (defined as before) w.r.t. $y^{p'}_{\boldsymbol{\beta}^{p'}}$ gives

$$\begin{aligned} \mathrm{d}y^{p'}_{\boldsymbol{\beta}^{p'}} &= \frac{\partial l}{\partial y^{p'}_{\boldsymbol{\beta}^{p'}}} = \sum_{\mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f} \frac{\partial l}{\partial f_{\boldsymbol{\alpha}}} \frac{\partial f_{\boldsymbol{\alpha}}}{\partial y^{p'}_{\boldsymbol{\beta}^{p'}}} = \sum_{\mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \overline{f}}{\partial \overline{s}} \sum_{\substack{k \in \Psi \\ \widehat{A}^{p'}\widehat{\boldsymbol{\alpha}} = \boldsymbol{\beta}^{p'}}} \frac{\partial s}{\partial y^{p'}_{\widehat{A}^{p'}\widehat{\boldsymbol{\alpha}}}} \\ &= \sum_{\substack{\mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f \\ k \in \Psi \\ \widehat{A}^{p'}\widehat{\boldsymbol{\alpha}} = \boldsymbol{\beta}^{p'}}} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \overline{f}}{\partial \overline{s}} \frac{\partial s}{\partial y^{p'}_{\widehat{A}^{p'}\widehat{\boldsymbol{\alpha}}}} = \sum_{\widehat{\boldsymbol{\alpha}} \in \widehat{\Gamma}} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \widehat{f}}{\partial y^{p'}_{\widehat{A}^{p'}\widehat{\boldsymbol{\alpha}}}} \end{aligned} \tag{50}$$

with the "sum-liberated" scalar function

$$\widehat{f}(y^1_{\widehat{A}^1\widehat{\boldsymbol{\alpha}}}, \dots, y^{P'}_{\widehat{A}^{P'}\widehat{\boldsymbol{\alpha}}}, x^1_{A^1\boldsymbol{\alpha}}, \dots, x^P_{A^P\boldsymbol{\alpha}}) \triangleq \overline{f}\left(s(y^1_{\widehat{A}^1\widehat{\boldsymbol{\alpha}}}, \dots, y^{P'}_{\widehat{A}^{P'}\widehat{\boldsymbol{\alpha}}}), x^1_{A^1\boldsymbol{\alpha}}, \dots, x^P_{A^P\boldsymbol{\alpha}}\right) \tag{51}$$

and the "sum-extended" multi-index set

$$\widehat{\Gamma} \triangleq \left\{ \begin{bmatrix} \boldsymbol{\alpha} & k \end{bmatrix} \,\middle|\, \boldsymbol{\alpha} \in \mathbb{Z}^{D_f} \wedge k \in \Psi \wedge \mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f \wedge \widehat{A}^{p'} \begin{bmatrix} \boldsymbol{\alpha} & k \end{bmatrix} = \boldsymbol{\beta}^{p'} \right\} . \tag{52}$$

Note that (50) equals the original expression for the derivative (38) but with $\overline{f}$ replaced by $\widehat{f}$, which is the same as $\overline{f}$ but with the sum symbol removed, and Γ replaced by $\widehat{\Gamma}$, which additionally includes the conditions on k from the sum.

Thus handling summations can be done using the previously described strategy for derivation by extending it as follows. Each sum symbol (!) in the function f to be derived is removed, its summation index is appended to the multi-index $\boldsymbol{\alpha}$ of f and its summation range is included as an additional constraint in the set Γ. This process is iterated for nested sums. When indexing into $\mathrm{d}f$ the additional indices in $\boldsymbol{\alpha}$ introduced by sums are ignored.

5.3 Element-wise Derivation Algorithm

Algorithm 4 computes expressions for derivatives $\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \partial l / \partial x^p_{\boldsymbol{\beta}^p}$ of a element-wise defined function f. If an expression for the Jacobian $\partial f_{\boldsymbol{\alpha}'} / \partial x^p_{\boldsymbol{\beta}^p}$ is desired, it can be obtained from $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$ by substituting

$$\mathrm{d}f_{\boldsymbol{\alpha}} \triangleq \prod_d \delta_{\alpha_d, \alpha'_d} \,.$$

Since the summation ranges (45) in a produced derivative are of the same form as the index ranges (42) of the input function and we shown in section 5.2 how to handle summations in the input function, we can iteratively apply the derivation algorithm on derivative functions to obtain second and higher order derivatives. Therefore the set of element-wise defined functions using linear combination of indices for its arguments is closed under the operation of derivation.

Algorithm 4: Element-wise expression derivation

Input: element-wise defined tensor-valued function f taking P tensor arguments $x^1, \dots, x^P$; expression of derivative $\mathrm{d}f_{\boldsymbol{\alpha}} \triangleq \partial l / \partial f_{\boldsymbol{\alpha}}$

Output: expression of derivatives w.r.t. arguments $\mathrm{d}x^p_{\boldsymbol{\beta}^p} \triangleq \partial l / \partial x^p_{\boldsymbol{\beta}^p}$

```
1  for $p \in \{1, \dots, P\}$ do                                  // loop over arguments $x^p$
2      $\mathrm{d}x^p_{\boldsymbol{\beta}^p} \longleftarrow 0$
3      for $q \in \{1, \dots, Q_p\}$ do                // loop over index expressions for $x^p$
           // compute derivative expression w.r.t. $x^p_{A^{pq}\boldsymbol{\alpha}}$ using reverse
              accumulation automatic differentiation (sec. 2)
4          $\delta \longleftarrow \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial f_{\boldsymbol{\alpha}}}{\partial x^p_{A^{pq}\boldsymbol{\alpha}}}$                // ignore sum symbols within $f$
           // compute range constraints from shape of $f$ and limits of
              occurring sums
5          $\Omega \longleftarrow \{\text{range constraints on } \boldsymbol{\alpha} \text{ of the form } R\boldsymbol{\alpha} \geq r\}$
           // compute Smith normal form (sec. 4.1) to obtain the following
6          $I \longleftarrow$ integer pseudo-inverse of $A^{pq}$
7          $K \longleftarrow$ integer kernel of $A^{pq}$
8          $C \longleftarrow$ integer cokernel of $A^{pq}$
           // rewrite constraints using $\boldsymbol{\beta}^p$ and kernel factors $\boldsymbol{z}$
9          $\Omega' \longleftarrow \{R K \boldsymbol{z} \geq r - R I \boldsymbol{\beta}^p \mid (R\boldsymbol{\alpha} \geq r) \in \Omega\}$
           // solve $\Omega'$ for $\boldsymbol{z}$ using Fourier-Motzkin elimination (sec. 4.2)
10         $\Sigma \longleftarrow \{\text{range constraints } \Omega' \text{ on } \boldsymbol{z} \text{ transformed into form (45)}\}$
           // generate derivative expressions
11         $\mathrm{d}x^p_{\boldsymbol{\beta}^p} \longleftarrow \mathrm{d}x^p_{\boldsymbol{\beta}^p} + \delta_{C\boldsymbol{\beta}^p} \sum_{\boldsymbol{z} \in \Sigma} \delta|_{\boldsymbol{\alpha} = I\boldsymbol{\beta}^p + K\boldsymbol{z}}$        // use form (47) for sum
```

6 Example and Numeric Verification

The implementation code is provided at https://github.com/surban/TensorAlgDiff. In our implementation and thus in this example we use zero-based indexing, i.e. a vector $x \in \mathbb{R}^N$ has indices $\{0, \dots, N-1\}$, as it is usual in modern programming languages. Given the function

$$f_{ij}(a, b, c, \boldsymbol{d}) = \exp\left[-\sum_{k=0}^{4} \left((a_{ik} + b_{jk})^2 \, c_{ii} + d_{i+k}^3\right)\right]$$

where the shapes of the arguments are $a \in \mathbb{R}^{3\times5}$, $b \in \mathbb{R}^{4\times5}$, $c \in \mathbb{R}^{3\times3}$ and $d \in \mathbb{R}^8$ and the shape of f is $f \in \mathbb{R}^{3\times4}$ the derivation algorithm produces the following output:

```
Input: f[i; j] = exp (-sum{k}_0^4 (((a[i; k] + b[j; k]) ** 2 * c[i; i] + d[i + k] ** 3)))
Derivative of f wrt. a: da[da_0; da_1] = sum{da_z0}_0^3 (((-(df[da_0; da_z0] * exp (-sum{k}
    _0^4 (((a[da_0; k] + b[da_z0; k]) ** 2 * c[da_0; da_0] + d[da_0 + k] ** 3))))) * c[
    da_0; da_0] * 2 * (a[da_0; da_1] + b[da_z0; da_1]) ** (2 - 1)))
Derivative of f wrt. b: db[db_0; db_1] = sum{db_z0}_0^2 (((-(df[db_z0; db_0] * exp (-sum{k}
    _0^4 (((a[db_z0; k] + b[db_0; k]) ** 2 * c[db_z0; db_z0] + d[db_z0 + k] ** 3))))) * c[
    db_z0; db_z0] * 2 * (a[db_z0; db_1] + b[db_0; db_1]) ** (2 - 1)))
Derivative of f wrt. c: dc[dc_0; dc_1] = if {dc_0 + -dc_1 = 0} then (sum{dc_z1}_0^4 (sum{
    dc_z0}_0^3 (((a[dc_1; dc_z1] + b[dc_z0; dc_z1]) ** 2 * (-(df[dc_1; dc_z0] * exp (-sum{
    k}_0^4 (((a[dc_1; k] + b[dc_z0; k]) ** 2 * c[dc_1; dc_1] + d[dc_1 + k] ** 3)))))))))
    else (0)
Derivative of f wrt. d: dd[dd_0] = sum{dd_z1}_(max [0; -2 + dd_0])^(min [4; dd_0]) (sum{
    dd_z0}_0^3 (((-(df[dd_0 + -dd_z1; dd_z0] * exp (-sum{k}_0^4 (((a[dd_0 + -dd_z1; k] + b
    [dd_z0; k]) ** 2 * c[dd_0 + -dd_z1; dd_0 + -dd_z1] + d[dd_0 + -dd_z1 + k] ** 3))))) *
    3 * d[dd_0] ** (3 - 1))))
```

The operator `**` denotes exponentiation in this output. The Kronecker delta has been encoded as a "*if x then y else z*" expression for more efficiency. Internally these expressions are represented as graphs, thus subexpressions occurring multiple times are only stored and evaluated once and no expression blowup as with symbolic differentiation occurs. To cleanup the generated expressions from the automatic differentiation algorithm an expression optimization step, which pre-evaluates constant parts of the expressions, should be incorporated. However, since this is not part of the core derivation problem, it has not been performed for this demonstration. These derivative expressions have been verified by using random numeric values for the arguments and comparing the resulting values for the Jacobians with results from numeric differentiation.

7 Conclusion

We have presented a method to compute symbolic expressions for derivatives of element-wise defined tensor-valued functions. These functions may contain summations and the indices of its arguments can be an arbitrary linear combination of the function indices. The output of our algorithm is an explicit symbolic expression for each element of the derivative. Thus the resulting expressions are very well suited for massively parallel evaluation in a lock- and synchronization-free CUDA kernel, which computes one element of the derivative per thread. No temporary memory is necessary for the evaluation of the derivatives. The derivatives themselves may contain additional summations over indices which have become free in the derivative. The output of the algorithm specifies the ranges of these sums as a maximum or minimum over a set of linear combinations of the derivative indices; therefore computing the numerical range at evaluation time costs only two matrix multiplications per loop run (not iteration).

References

[Adkins and Weintraub, 1999] Adkins, W. A. and Weintraub, S. H. (1999). *Algebra: An Approach via Module Theory (Graduate Texts in Mathematics)*. Springer.

[Coppersmith and Winograd, 1987] Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM.

[Dantzig, 2016] Dantzig, G. (2016). *Linear programming and extensions*. Princeton university press.

[Dantzig and Eaves, 1973] Dantzig, G. B. and Eaves, B. C. (1973). Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297.

[Dantzig and Thapa, 2006a] Dantzig, G. B. and Thapa, M. N. (2006a). *Linear programming 1: introduction*. Springer Science & Business Media.

[Dantzig and Thapa, 2006b] Dantzig, G. B. and Thapa, M. N. (2006b). *Linear programming 2: theory and extensions*. Springer Science & Business Media.

[Farzad, 2013] Farzad (2013). Cuda atomic operation performance in different scenarios.

[Kjolstad et al., 2017] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. (2017). The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29.

[Knuth, 1997] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Nvidia, 2017] Nvidia (2017). *NVIDIA CUDA C Programming Guide*, version 9.0 edition.

[Smith, 1860] Smith, H. S. (1860). On systems of linear indeterminate equations and congruences. *Proceedings of the Royal Society of London*, 11:86–89.

[Strassen, 1969] Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356.

[yidiyidawu, 2012] yidiyidawu (2012). Cuda performance of atomic operation on different address in warp.