Symbolic Tensor Differentiation for Applications in Machine Learning

Andrei Zhabinski
Belarussian State University
of Informatics and Radioelectronics
Minsk, Belarus
Email: andrei.zhabinski@gmail.com

Sergey Zhabinskii
Belarussian State University
of Informatics and Radioelectronics
Minsk, Belarus
Email: sergey.zhabinskii@gmail.com

Dzmitry Adzinets
Belarussian State University
of Informatics and Radioelectronics
Minsk, Belarus
Email: adzinets@bsuir.by

***Abstract*—Automated methods for computing derivatives of cost functions are essential to many modern applications of machine learning. Reverse-mode automatic differentiation provides relatively cheap means for it but generated code often requires a lot of memory and is hardly amenable to later optimizations. Symbolic differentiation, on the other hand, generates much more flexible code, yet applying it to multidimensional tensors is a poorly studied topic. In this paper, we present a method for symbolic tensor differentiation based on extended Einstein indexing notation, which allows to overcome many limitations of both - automatic and classic symbolic differentiation, and generate efficient code for CPU and GPU.**

Keywords: symbolic differentiation, Einstein notation, machine learning.

I. Introduction

A significant portion of machine learning (ML) algorithms directly relies on gradient-based optimization methods. As their name states, these methods require computing a gradient of a loss function on each step of optimization. Simple models like logistic regression have well-known formulas for computing partial derivatives. However, recent progress in machine learning and especially deep neural networks has given a rise to much more complicated models and loss functions.

Manually computing gradients of such functions is time-consuming and error-prone, so often computer-based methods are used instead. Such methods fall into several categories, and each category has their advantages and disadvantages. In this paper, we present a method that combines parts of 2 of these categories and is specifically designed with a focus on machine learning applications. One important difference from other approaches is that our method also supports efficient calculation of derivatives of functions from R^m to R^n where both m and n are large, whereas other methods either require one of them to be small (e.g. [1], [2]), or can't handle vector functions at all (e.g. [3], [4]).

The rest of this paper is structured as follow. In the next section, we revisit major families of computer-based differentiation algorithms and explain where our method falls in. In section 3 we describe actual method as applied to scalars (numbers), while in section 4 we extend it to higher-order tensors using Einstein notation.

II. Overview of computer-based differentiation methods

To the best of our knowledge, most computer-based differentiation methods are divided into 3 main categories:

1) Numeric (also known as finite difference methods)
2) Symbolic
3) Automatic

Numeric differentiation (ND) methods are the simplest ones and directly rely on the definition of derivative. We say that $\frac{df}{dx}$ is a derivative of a function $f(x)$ with respect to x if:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Given this definition, we can calculate derivative of any function by computing it at x and $x + \Delta x$ where Δx is some little constant, and then dividing the result by that constant.

The major advantage of this method is that it's able to compute derivatives of any function differentiable at both - x and Δx. The main disadvantage comes from complexity of choosing good Δx: too small values may lead to round-off errors during floating point operations, while too large values lead to weak approximation of the real value of a derivative. Attempts have been done to improve precision using multiple differences (e.g. [5]), but in general case accuracy of numeric differentiation stays unstable.

Symbolic differentiation (SD) takes a different approach - instead of evaluating a function, it constructs a symbolic expression representing its derivative. Technically, symbolic differentiaton relies on a fact that algebraic expressions are either primitive (e.g. summation, product, exponent, etc.) or a combination of the above. For primitive expressions there's already a well-known set of rules, i.e. $\frac{d}{dx} cos(x) = sin(x)$. Combined expressions are handled using the chain rule, i.e. if $z = z(y)$ is a function of y which in turn is a function $y = y(x)$ of x, then the derivative $\frac{dz}{dx}$ may be expressed as:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Or, in case of several intermediate functions y_i:

$$\frac{dz}{dx} = \sum_i \frac{dz}{dy_i}\frac{dy_i}{dx}$$

The chain rule and differentiation rules for primitive expressions make it possible to divide any algebraic expression into primitive parts, differentiate each part and then combine the results. Here we demonstrate this method by example.

Let's say we want to differentiate expression $f(x) = sin(x^2)$. Since there are only 2 calls, we can introduce one intermediate function $g(x)$ so that $f(x) = sin(g(x))$. Using the chain rule from above we get:

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx} = \frac{d(sin(g))}{dg}\frac{d(x^2)}{dx}$$

From primitive rules we know that $\frac{d}{dg}sin(g) = cos(g)$ and $\frac{d}{dx}x^2 = 2x$. Replacing g with its original value and multiplying two derivatives we get:

$$\frac{df}{dx} = cos(x^2) \times 2x$$

Which is the correct derivative of the specified function.

Unlike numeric approach, symbolic differentiation produces code for *exact* calculation of derivatives. Also, in many cases this code is as efficient as possible and close to what a human expert would derive. Moreover, produced symbolic expression may then be further optimized (e.g. to fit memory requirements or improve numeric stability) or even translated to GPU or any other computation engine, which is extremely valuable property in the context of neural networks and related models.

On the other hand, symbolic differentiation has much more limited applicability. First of all, since the goal of this family of methods is to produce symbolic expression of a derivative, it is limited to functions continuously differentiable in all points. In computer program, however, most conditions and loops introduce discontinuity, so they are normally not allowed. Another restriction of symbolic differentiation is that most practical implementations (including highly adopted [3] and [4]), are essentially univariate, i.e. they don't support vector inputs and are thus inapplicable to most machine learning functions.

Automatic differentiation (AD) is a set of techniques to numerically evaluate the derivative of a function. Similarly to symbolic differentiation, it decomposes algebraic expressions into a set of primitive subexpressions and then uses the chain rule to combine the results. Unlike symbolic approach, however, AD doesn't produce a new expression, but only computes a derivative at a point, so it can easily handle conditions and loops. AD has 2 main modes: forward and reverse.

Forward-mode AD consists of a single pass from input arguments to the value of a function, during which both - value of the function and its derivative - are calculated. Typically, this mode is implemented using dual numbers - a special extension of real numbers with additional part that holds the derivative of a variable in question. However, this method has pretty bad complexity for function $R^m \to R^n$ where $m \gg n$ such as most loss functions in machine learning. Thus we don't dive deeper in the description of this method and guide interested reader to [2].

Reverse-mode AD consists of 2 passes. On the *forward* pass it computes values of all primitive expressions and records them into intermediate variables on a so-called "tape". On the *reverse* pass, the method starts with the derivative of output variable by itself (i.e. always 1) and traverses expression tree backward to calculate derivatives of the output w.r.t. each of intermediate variables. Here's an example of such computation.

Let's say we want to differentiate function $z = sin(x^2)$ at point $x = 2$. This complex expression contains 2 primitive expressions, which we compute and immediately write the results onto the tape:

1) $y = x^2 = 4$
2) $z = sin(y) \approx -0.756$

By convention, we also replace input and output variables by new indexed names to make all variables look similar. This way we get:

1) $w_1 = x = 2$
2) $w_2 = w_1^2 = 4$
3) $w_3 = sin(w_2) \approx -0.756$

We seek to find derivative $\frac{dz}{dx} = \frac{dw_3}{dw_1}$. We start with the fact that the derivative of the output variable w.r.t. itself is $\frac{dw_3}{dw_3} = 1$ (the proof is trivially inferred from the definition of derivative).

From the chain rule we know that $\frac{dw_3}{dw_2} = \frac{dw_3}{dw_3} \times \frac{dw_3}{dw_2}$. We already know that $\frac{dw_3}{dw_3} = 1$ and from primitive rules we also know that $\frac{d(sin(w_2))}{dw_2} = cos(w_2) = cos(4) \approx -0.654$, so multiplying these parts we get $\frac{dw_3}{dw_2} = 1 \times 0.654 = 0.654$. Note, that AD doesn't ever create or transform symbolic expressions, but just calculates actual values of intermediate variables and their derivatives.

Reverse-mode AD has gained particular popularity in the field of machine learning and is implemented in such frameworks as Theano [6] and TensorFlow [7], as well as in dedicated libraries such as AutoGrad [8]. Strengths of this method include efficient (in terms of processing time) calculation of derivative of functions $R^m \to R^n$ where $m \gg n$ and transparent handling of multivariate vectors. Downsides of AD include high memory usage (because of the need to keep all variables on a tape during both - forward and reverse pass) and lack of symbolic representation that could be used for further optimizations and code generation.

There's also a few methods that try to combine strengths of symbolic and automatic differentiation. One notable example is **D*** algorithm from [9]. In their paper, Guenter et al. describe a library in C++ that overloads common algebraic operations for their *V(ariable)* class to produce expression graph, and then aggressively optimize this graph to eliminate repeating blocks. **D*** allows for vector-valued functions where each output component may represent an independent function of input arguments. This gives flexibility in defining vector-valued functions, but in the worst case (i.e. when there are no repeating blocks to be eliminated) may result in a run time proportional to the number of output values.

Our method is similar in spirit to **D***, but instead of supporting independent functions, we limit output components to functions of the same basic structure and only different indices (e.g. $z_i = f(x_j, y_k)$). This is much more common in machine learning and enables generating more compact code with much smaller run time. Moreover, our method works with functions that output not only vectors, but also matrices and, in general, tensors of any rank.

In the next section we describe our method in a way that works for scalars, and in section 4 we extend this framework to support higher-order tensors.

III. Symbolic method for scalars

The ultimate goal of our method is to be able to produce symbolic derivatives for algebraic functions commonly used in machine learning. Since we seek for a symbolic representation, we limit our framework to continuously differentiable function, i.e. we don't support conditional operators and loops as they may, in general case, introduce points of discontinuity. To our mind, this is not a hard limitation in the context of ML since most loss functions that use loops can be transformed into a form with summation. We do, however, support operators like min, max, $sign$ and similar that may be thought of as conditional, but don't create discontinuity points.

In general, our method is based on approach used in reverse-mode AD, but instead of computing *values* of variables and derivatives, we compose their *symbolic expressions* to build a computational graph suitable for further optimization. For demonstration purposes, let's consider expression $z = x_1 x_2 + sin(x_1)$ and find derivatives $\frac{dz}{dx_1}$ and $\frac{dz}{dx_2}$.

Forward pass in our case consists of 2 stages: recording all primitive operations onto a tape and evaluating "example values" (which we explain later in this section). Recording operations includes parsing subexpressions into intermediate variables and constructing a list of primitive expressions, e.g. for expression above such a list would look like this:

1) $w_1 = x_1$
2) $w_2 = x_2$
3) $w_3 = w_1 \times w_2$
4) $w_4 = sin(w_1)$
5) $z = w_5 = w_3 + w4$

This expression list, however, lacks type information needed to distinguish between scalars and higher-rank tensors. To obtain this information, we introduce a notion of "example values" - values attached to each variable and having the same type as real values could have. E.g. we can use value 1 for x_1 and x_2 to indicate that input parameters are scalars, or random 2x2 matrices to indicate that they are instances of tensor-2. Note, that real values passed to generated expression later don't need to have the same value or size (for tensors), but only the same type and number of dimensions.

After we have obtained example values of input parameters we can evaluate each primitive expression and infer example values for all intermediate and final variables.

Reverse pass, just as in the case of AD, starts with the derivative of an output variable w.r.t. to itself and propagates derivatives back to input variables. However, instead of propagating concrete values we *aggregate symbolic expressions*, i.e. build a formula describing how to calculate it from input and intermediate variables. We demonstrate it by example:

1) $\frac{dz}{dw_5} = \frac{dw_5}{dw_5} = 1$
2) $\frac{dz}{dw_4} = 1 \times 1 = 1$
3) $\frac{dz}{dw_3} = 1 \times 1 = 1$
4) $\frac{dz}{dw_1} = 1 \times cos(w_1) = cos(w_1)$
5) $\frac{dz}{dw_2} = 1 \times w_1$
6) $\frac{dz}{dw_1} = cos(w_1) + 1 \times w_2 = cos(w_1) + w_2$

Note that in our calculations we encounter $\frac{dz}{dw_1}$ twice - at lines 4 and 6. First time we calculate derivative of $\frac{dz}{dw_1}$ via w_4 only, while second time we also include the derivative via w_3. Also note, that we simplify expressions where possible, e.g. use 1 instead of 1×1 - although it's not strictly necessary, it simplifies reading and reduces run time of implementation.

So we finish with 2 formulas:

$$\frac{dz}{dx_1} = \frac{dz}{dw_1} = cos(w_1) + w_2$$

$$\frac{dz}{dx_2} = \frac{dz}{dw_2} = w_1$$

These formulas may then be applied to any (scalar) input values x_1 and x_2 to immediately obtain the result.

IV. Symbolic method for higher-order tensors

It's tempting to apply the same approach to differentiation of higher-order derivatives such as vectors and matrices. For example, for dot product of two vectors $z = \boldsymbol{x} \cdot \boldsymbol{y}$ we can easily show that $\frac{dz}{d\boldsymbol{x}} = \boldsymbol{y}$. However, this approach doesn't work for more complex operations because not all such derivatives have symbolic representation. One such example is the matrix-by-vector product:

$$\boldsymbol{W}\boldsymbol{x}$$

In this case derivative $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}}$ is a tensor with 3 dimensions that cannot be straightforwardly expressed as a combination of input vectors and matrices.

Although we cannot infer symbolic rules for tensors themselves, we still can do it for tensor components. We note here that for any function on tensors each component of an output tensor is a function of zero or more components of input tensors. For example, matrix-by-vector product above can be rewritten as:

$$\sum_k W_{ik} x_k, \quad \forall i$$

Now instead of a tensor expression, we have a set of scalar expressions over indexed variables W_{ik} and x_k, which is much easier target. However, before we move to differentiation of these indexed expression, we need to make one more change to out notation.

Since we want to use symbolic differentiation from a programming language, we need a way to represent such expressions in programming notation, and symbols like $\sum$ and $\forall$ don't have straightforward counterpart in programming. Even if we find appropriate representation, tensor expressions in index notation very quickly become bulky. Fortunately, these issues have already been addressed previously and have a solution in so-called Einstein indexing notation or just **Einstein notation**.

According to classic Einstein notation [10], unless otherwise stated:

1) a summation is assumed over all indices that appear twice in a product
2) no summation is assumed over indices that appear only once

Given these rules, we can rewrite matrix-by-vector product in a shorter form:

$$W_{ik}x_k$$

which also plays well with array indexing notation in many programming languages, e.g. in Python or Julia:

```
W[i,k] * x[k]
```

Note, that for an index to be a "sum" index it should appear exactly in the same *product term*. For example, in the following expression index i, although appearing twice, is still "for all" index:

$$x_i + y_i$$

The "unless otherwise stated" part is also important. In particular, when an expression is an equation, the common approach is to consider all indices appearing on the left-hand side (LHS) to be "for all" and all the others - "sum" indices. E.g. we may write:

$$y = x_i$$

to indicate summation over components of vector $\mathbf{x}$.

Although Einstein notation is quite powerful by itself, it was designed for humans and doesn't cover a few important cases that may arise during automatic expression processing. Below we describe these cases and introduce several extensions to classic Einstein notation that help to overcome them.

First of all, since summation is a special operation that requires either product term or an equation with LHS, how do we represent it when none of these conditions is available? For example, how do we express:

$$\sum_i x_i$$

To deal with such cases we introduce a notion of **pseudo-one** - a special object similar to a tensor of all ones, but with undefined size. For example, we write one-dimensional pseudo-one as:

$$1_i$$

This object is only valid when it's being multiplied by other variables, and in this case it simply results in summation over pseudo-one indices. Now, having a notion of pseudo-one, we can represent a sum of vector components as:

$$\sum_i x_i \to x_i 1_i$$

Another corner case is conditional expressions. For example, as we will show later, in expression $y_i = W_{ik}x_k$ derivative $\frac{\partial y_i}{\partial W_{mn}}$ is a 3 dimensional tensor with components defined as:

$$\frac{\partial y_i}{\partial W_{mn}} = \begin{cases} x_n, & \text{if } i = m, \\ 0, & \text{otherwise.} \end{cases}$$

Again, this expression is pretty inconvenient to work with, especially when we have to multiply it by other expressions of the same kind. To tackle with it, we introduce **guards** - conditional subexpressions on indices that either pass the main expression as is (if the condition is true) or turn it into zero (otherwise). For example, derivative $\frac{\partial y_i}{\partial W_{mn}}$ can now be written as:

$$\frac{\partial y_i}{\partial W_{mn}} = x_n \quad |i = m$$

where everything after a bar is a guard.

One may argue that this notation is limited because it assumes one of conditional branches to be zero and it's not possible to represent, for example, such expression:

$$Z_{imn} = \begin{cases} x_n, & \text{if } i = m, \\ y_n, & \text{otherwise.} \end{cases}$$

Indeed, notation with guards is *optimized* for conditions with zero branch, but one can still represent other conditions as a sum of separate branches multiplied by postitive or negative condition. E.g.:

$$Z_{imn} = (x_n | i = m) + (y_n | i \neq m)$$

Now, having this extended Einstein notation in mind, we can eventually describe our method for symbolic tensor differentiation.

Our method consists of 5 principal steps:

1) parse expression into a list of primitive expressions
2) convert each primitive expression to Einstein notation
3) apply known differentiation rules to primitive expressions
4) combine resulting derivatives using the chain rule
5) convert derivative expression from Einstein notation to vectorized one

We do **expression parsing** exactly the same way as we did previously: each function call is transformed into a separate primitive expression. For example, linear transformation:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

is parsed into two simpler expressions:

$$\boldsymbol{t} = \boldsymbol{W}\boldsymbol{x}$$
$$\boldsymbol{y} = \boldsymbol{t} + \boldsymbol{b}$$

We **convert from vectorized into Einstein notation** using a set of predefined rules. Each rule simply describes how components of output tensor depend on components of input tensors. Since at this step we deal only with primitive expressions, we only need to define one rule for each operation and operand types plus one special rule for elementwise operations (e.g. Hadamard product, which we denote here with $\circ$). The rules are easy to infer, so we provide here only a few examples for demonstration purposes:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} \quad \rightarrow \quad y_i = W_{ik}x_k$$
$$\boldsymbol{Z} = \boldsymbol{X} \circ \boldsymbol{Y} \quad \rightarrow \quad Z_{ij} = X_{ij} \times Y_{ij}$$
$$y = sum(\boldsymbol{x}) \quad \rightarrow \quad y = x_i \times 1_i$$

Rules for finding **derivatives of primitive expressions in Einstein notation** are similar to those of scalar expressions, but with two exceptions:

1) all variables are indexed, and
2) where derivative turns zero, guards are used

We designate tensor derivatives in Einstein notation as:

$$\frac{\partial Y_I}{\partial X_J}$$

where Y is the dependent variable, X is the variable we differentiate with respect to, and I and J are sets of unique indices of appropriate length. For example, given an expression:

$$y_i = W_{ik}x_k$$

derivative $\frac{\partial y_i}{\partial W_{mn}}$ describes how ith component of output vector $\boldsymbol{y}$ depends on (m, n)th component of input matrix $\boldsymbol{W}$. It's easy to show that y_i depends only on ith row of matrix $\boldsymbol{W}$, i.e. derivative is non-zero only when $i = m$, and when this condition holds, changing W_{mn} by 1 leads to corresponding change in y_i by x_n. We write both these statements as:

$$\frac{\partial y_i}{\partial W_{mn}} = x_n \quad |i = m$$

Other rules are inferred in a similar way, so we show only a couple of them for reference:

$$y_i = W_{ik}x_k \quad \rightarrow \quad \frac{\partial y_i}{\partial x_m} = W_{im}$$
$$Z_{ij} = X_{ij} \times Y_{ij} \quad \rightarrow \quad \frac{\partial Z_{ij}}{\partial X_{mn}} = 1 \quad |i = m, j = n$$
$$y = x_i \times 1_i \quad \rightarrow \quad \frac{\partial y}{\partial x_m} = 1$$

We **combine derivatives of primitive expressions** using the standard chain rule. If guards are present in one or both expressions, they are combined together. For example, if we have derivatives of 2 primitive expressions in the same operator chain:

$$\frac{\partial Z_{ij}}{\partial Y_{mn}} = U_{nj} \quad |i = m$$
$$\frac{\partial Y_{mn}}{\partial X_{st}} = W_{tn} \quad |m = s$$

then after applying the chain rule we get:

$$\frac{\partial Z_{ij}}{\partial X_{st}} = U_{nj}W_{tn} \quad |i = m, m = s$$

which is naturally simplified to:

$$\frac{\partial Z_{ij}}{\partial X_{st}} = U_{nj}W_{tn} \quad |i = s$$

It's also worth to note that despite automatic construction the resulting expression is still valid in Einstein notation: we indeed sum out n index and leave all the others.

Finally, we **convert expressions in Einstein notation back to vectorized one** using predefined rules. Note, that conversion between vectorized and Einstein notation doesn't make one-to-one relation, but rather many-to-one (in both directions). For example, both of the following rules are used if different contexts:

$$y = x_i 1_i \quad \rightarrow \quad y = sum(\boldsymbol{x})$$
$$y = x_i \quad \rightarrow \quad y = sum(\boldsymbol{x})$$

Both expressions are valid and both designate sum over components of vector $\boldsymbol{x}$.

V. Code generation for GPU

Conversion to vectorized notation may be seen as code generation for executing on CPU (central processing unit). But it's not the only possible backend - we can equally generate code for alternative runtime systems. One popular choice of such systems is GPU (graphics processing unit) which provides means for executing code with very high level of parallelism.

In a typical GPGPU (general-purpose GPU) program, data is copied into device's memory, and then a number of so-called kernels is run over them. Each kernel consists of a set of relatively simple instructions executed on each datum in parallel (up to the number of available GPU threads). This plays especially well with our Einstein notation which essentially is a way to describe how to calculate each output variable.

Below we describe several techniques for code generation for GPU that we used in our work.

One of the major bottlenecks in GPU computing is **data transfer** between main and device's memory, so we try to avoid it as much as possible. Fortunately, in our expressions,

we always know input and output variables, so we load them to and from a device exactly once for each function we analyze.

Another advantage of upfront expression analysis is that we can **reuse data buffers**. To do this, we keep track of:

1) type and size of each variable's buffer
2) variable's lifetime

Each time we need to create a new temporary variable we first look for a suitable buffer in a cache of existing variables. If there's one and the corresponding variable isn't used later than current expression, this buffer is reused. One very frequent example is elementwise operations. E.g. the following expression (in mathematical notation):

$$c_i := a_i + b_i$$

is actually translated into

$$b_i := a_i + b_i$$

(Here we use := to denote assignment in order not to abuse equality sign =)

Another useful property of elementwise operations is that they may be **grouped into a single kernel**. Enqueuing a kernel has its own overhead, running several instructions in a single kernel helps to fix it. For example, in our tests running a logistic function $1/(1 + exp(-x_i))$ in single kernel turned to be 1.4 times faster than running 3 separate kernels for exponent, summation and division (0.057 ms vs 0.08 ms for 50000-element 32-bit vectors using OpenCL).

Finally, for matrix multiplication and other aggregation functions we use a library implementing **BLAS** specification [11].

VI. Implementation in Julia

We provide a reference implementation in *XDiff.jl* [12] package in the Julia programming language. Below we provide the rationale for choosing the language and some other design decisions.

We chose Julia for several main reasons:

1) Julia supports literals for vectors and matrices and a large number of built-in mathematical functions
2) it also supports symbolic programming, i.e. variables and expressions may be represented as first-class citizens in Julia
3) operator overloading allows extendng semantics of expressions to support, for example, symbolic product and summation

For comparison, in Python/C++ TensorFlow the whole new language with its own abstract syntax tree and a set of standard functions had to be designed for the same purposes.

Below we describe how we transfer some concepts from mathematical to programming notation.

As we've already described in the text, we represent variable indices using standard array indexing, e.g.:

```
y[i] = W[i,k] * x[k]
```

Since Julia doesn't support indexing of number literals, we replace pseudo-one notation 1_n with symbol *I[n]*.

Finally, to represent guards we use index comparison (e.g. *(i == m)*) and add it as an additional product term to the main expression. E.g.

$$\frac{\partial y_i}{\partial W_{mn}} = x_n \quad |i = m$$

is translated into

```
dy_dW[i,m,n] = x[n] * (i == m)
```

When being multiplied by a number, boolean expression *(i == m)* is converted into either 1 (true) or 0 (false). Thus when the condition holds, guard simply disappears, otherwise the whole expression turns into zero.

We generate code for GPU in OpenCL's format. This choice is due to OpenCL's widespread and portability, however, described techniques can be equally applied to other GPGPU systems (notably, CUDA).

VII. Conclusion

We have introduced a method for symbolic tensor differentiation using extended Einstein notation. The proposed method can be efficiently applied to functions where both - input and output - are tensors. We have also described techniques for code generation for CPU and GPU. The correctness of approach is proved by our reference implementation.

References

[1] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," *arXiv:1607.07892 [cs.MS]*, 2016.

[2] B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt, "The stan math library: Reverse-mode automatic differentiation in C++," *CoRR*, vol. abs/1509.07164, 2015.

[3] Wolfram Research, Inc., "Mathematica 8.0," 2010.

[4] SymPy Development Team, *SymPy: Python library for symbolic mathematics*, 2016.

[5] B. Fornberg, "Generation of finite difference formulas on arbitrarily spaced grids," *Mathematics of Computation*, vol. 51, no. 184, pp. 699–706, 1988.

[6] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[8] D. Maclaurin, D. Duvenaud, M. Johnson, and R. P. Adams, "Autograd: Reverse-mode differentiation of native Python," 2015.

[9] B. Guenter, "Efficient symbolic differentiation for graphics applications," *ACM Trans. Graph.*, vol. 26, July 2007.

[10] K. Dullemond and K. Peeters, "Introduction to tensor calculus," 1991.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, pp. 308–323, Sept. 1979.

[12] A. Zhabinski, "Xdiff.jl," 2017.