JMLR: Workshop and Conference Proceedings (2014) 1–7 ICML 2014 AutoML Workshop

Automatic Differentiation of Algorithms for Machine Learning

Atılım Güneş Baydin ATILIMGUNES.BAYDIN@NUIM.IE
Barak A. Pearlmutter BARAK@CS.NUIM.IE
Department of Computer Science & Hamilton Institute
National University of Ireland Maynooth, Co. Kildare, Ireland

Abstract

Automatic differentiation—the mechanical transformation of numeric computer programs to calculate derivatives efficiently and accurately—dates to the origin of the computer age. Reverse mode automatic differentiation both antedates and generalizes the method of backwards propagation of errors used in machine learning. Despite this, practitioners in a variety of fields, including machine learning, have been little influenced by automatic differentiation, and make scant use of available tools. Here we review the technique of automatic differentiation, describe its two main modes, and explain how it can benefit machine learning practitioners. To reach the widest possible audience our treatment assumes only elementary differential calculus, and does not assume any knowledge of linear algebra.

Keywords: Automatic Differentiation, Machine Learning, Optimization

1. Introduction

Many methods in machine learning require the evaluation of derivatives. This is particularly evident when one considers that most traditional learning algorithms rely on the computation of gradients and Hessians of an objective function, with examples in artificial neural networks (ANNs), natural language processing, and computer vision (Sra et al., 2011).

Derivatives in computational models are handled by four main methods: (a) working out derivatives manually and coding results into computer; (b) numerical differentiation; (c) symbolic differentiation using computer algebra; and (d) automatic differentiation.

Machine learning researchers devote considerable effort for the manual derivation of analytical derivatives for a novel model they introduce, subsequently using these in standard optimization procedures such as L-BFGS or stochastic gradient descent. Manual differentiation has the advantage of avoiding approximation errors and instability known to be present in numerical differentiation, but can be prone to error and labor intensive. Symbolic computation methods address weaknesses of both manual and numerical methods, but often result in complex and cryptic expressions plagued with the problem of "expression swell".

The fourth technique, automatic differentiation (AD)[1] works by systematically applying the chain rule of calculus at the elementary operator level. AD allows accurate evaluation of derivatives with only a small constant factor of overhead and ideal asymptotic efficiency. Unlike the need for arranging algorithms into monolithic mathematical expressions for symbolic differentiation, AD can be applied to existing code with minimal change. Owing to

1. Also called "algorithmic differentiation" and less frequently "computational differentiation".

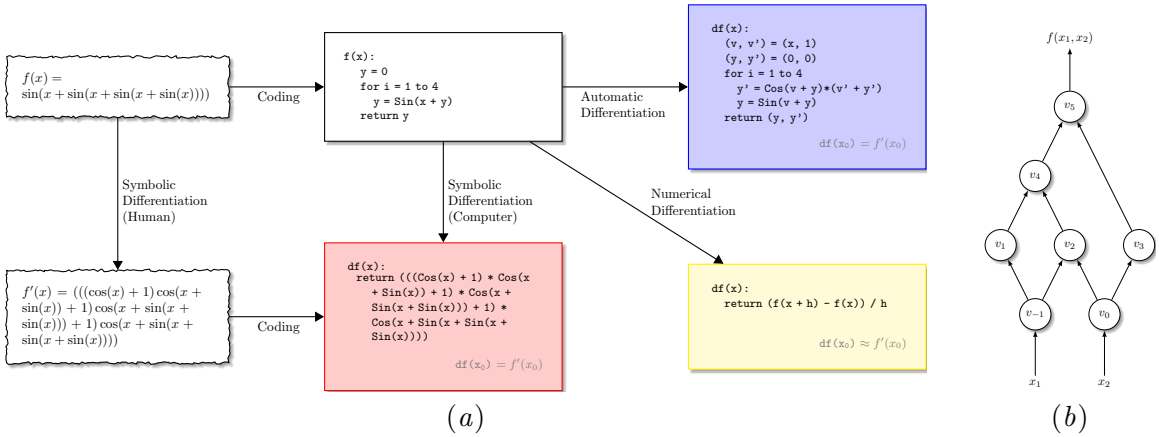

Figure 1: (a) Differentiation of mathematical expressions and code. Symbolic differentiation (lower center); numerical differentiation (lower right); AD (upper right). (b) Computational graph of the example $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$.

this, it is an established tool in applications such as real-parameter optimization (Walther, 2007), sensitivity analysis, and probabilistic inference (Neal, 2011).

Despite its widespread use in other fields, AD has been underused, if not unknown, by the machine learning community. How relevant AD can be for machine learning tasks is exemplified by the backpropagation method for ANNs, modeling learning as gradient descent in ANN weight space and utilizing the chain rule to propagate error values. The resulting algorithm can be obtained by transforming the network evaluation function through reverse mode AD. Thus, a modest understanding of the mathematics underlying the backpropagation method gives one already sufficient basis to grasp the technique.

Here we review AD from a machine learning perspective and bring up some possible applications in machine learning. It is our hope that the review will be a concise introduction to the technique for machine learning practitioners.

2. What AD Is Not

The term "automatic differentiation" has undertones that it is either symbolic or numerical differentiation. The output of AD is indeed numerical derivatives, while the steps in its computation do depend on algebraic manipulation, giving it a two-sided nature partly symbolic and partly numerical. Let us start by stressing how AD is different from, and in some aspects superior to, these two commonly encountered techniques (Figure 1 (a)).

AD is not numerical differentiation. Finite difference approximation of derivatives uses the original function evaluated at sample points. In its simplest form, it uses the standard definition $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$ and approximates the left-hand side by evaluating right-hand side with a small nonzero h. This is easy to implement, but inherently prone to truncation and round-off errors. Truncation tends to zero as $h \to 0$; however, at the same time, round-off increases and becomes dominant. Improvements such as higher-order finite differences or Richardson extrapolation do not completely eliminate approximation errors.

Table 1: Forward AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ at $(x_1, x_2) = (2, 5)$ and setting $\dot{x}_1 = 1$ to compute $\partial y / \partial x_1$.

Forward evaluation trace			Forward derivative trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + v_{-1} \times \dot{v}_0$	$= 1 \times 5 + 2 \times 0$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \cos v_0 \times \dot{v}_0$	$= \cos 5 \times 0$
v_4	$= v_1 + v_2$	$= 0.6931 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.6931 + 0.9589$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.6521$	$\dot{y}$	$= \dot{v}_5$	$= 5.5$

AD is not symbolic differentiation. One can generate exact symbolic derivatives through manipulation of expressions via differentiation rules such as $\frac{d}{dx}(u(x)v(x)) = \frac{du(x)}{dx}v(x) + u(x)\frac{dv(x)}{dx}$. This perfectly mechanistic process is realized in computer algebra systems such as Mathematica, Maple, and Maxima. Symbolic results can give insight into the problem and allow analytical solutions of optima (e.g. $\frac{df(x)}{dx} = 0$) in which case derivatives are no longer needed. Then again, they are not always efficient for run-time calculations, as expressions can get exponentially larger through differentiation ("expression swell").

3. AD Origins

For accurate numerical derivatives, it is possible to simplify symbolic calculations by only storing values of intermediate steps in memory. For efficiency, we can interleave, as much as possible, the differentiation and storage steps. This "interleaving" idea forms the basis of "Forward Accumulation Mode AD": apply symbolic differentiation to each elementary operation, keeping intermediate numerical results, in lockstep with the evaluation of the original function.

3.1. Forward Mode

All computations are ultimately compositions of a finite set of elementary operations with known derivatives. Combining derivatives of constituent operations through the chain rule gives the derivative of the overall composition. In Table 1, we have the example $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ represented as an *evaluation trace* of elementary operations—also called a Wengert list. Using the "three-part" notation of Griewank and Walther (2008), a trace of $f : \mathbb{R}^n \to \mathbb{R}^m$ is constructed from (a) $v_{i-n} = x_i, i = 1, \ldots, n$ input variables, (b) $v_i, i = 1, \ldots, l$ working variables, and (c) $y_{m-i} = v_{l-i}, i = m-1, \ldots, 0$ output variables. We can also represent a given trace of operations as a data flow graph, as shown in Figure 1 (b), which makes the dependency relations between intermediate variables explicit.

For computing the derivative with respect to, say x_1, we associate with each variable v_i a corresponding $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$. Applying the chain rule to each elementary operation in the forward trace, we generate the derivative trace on the right-hand side. Evaluating variables v_i one by one together with $\dot{v}_i$ gives us the required derivative in the final variable $\dot{v}_5$.

In general, for an $f : \mathbb{R}^n \to \mathbb{R}^m$ with n independent x_i as inputs and m dependent y_j as outputs, each forward pass of AD is initialized by setting the derivative of only one of

Table 2: Reverse AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ at $(x_1, x_2) = (2, 5)$. Setting $\bar{y} = 1$, $\partial y/\partial x_1$ and $\partial y/\partial x_2$ are computed in one reverse sweep.

Forward evaluation trace			Reverse adjoint trace			
v_{-1}	$= x_1$	$= 2$	$\bar{x}_1$	$= \bar{v}_{-1}$	$= 5.5$	
v_0	$= x_2$	$= 5$	$\bar{x}_2$	$= \bar{v}_0$	$= 1.7163$	
v_1	$= \ln v_{-1}$	$= \ln 2$	$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1(\partial v_1/\partial v_{-1})$	$= \bar{v}_{-1} + \bar{v}_1/v_{-1}$	$= 5.5$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2(\partial v_2/\partial v_0)$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.7163$
v_3	$= \sin v_0$	$= \sin 5$	$\bar{v}_{-1}$	$= \bar{v}_2(\partial v_2/\partial v_{-1})$	$= \bar{v}_2 \times v_0$	$= 5$
v_4	$= v_1 + v_2$	$= 0.6931 + 10$	$\bar{v}_0$	$= \bar{v}_3(\partial v_3/\partial v_0)$	$= \bar{v}_3 \times \cos v_0$	$= -0.2837$
v_5	$= v_4 - v_3$	$= 10.6931 + 0.9589$	$\bar{v}_2$	$= \bar{v}_4(\partial v_4/\partial v_2)$	$= \bar{v}_4 \times 1$	$= 1$
			$\bar{v}_1$	$= \bar{v}_4(\partial v_4/\partial v_1)$	$= \bar{v}_4 \times 1$	$= 1$
			$\bar{v}_3$	$= \bar{v}_5(\partial v_5/\partial v_3)$	$= \bar{v}_5 \times (-1)$	$= -1$
			$\bar{v}_4$	$= \bar{v}_5(\partial v_5/\partial v_4)$	$= \bar{v}_5 \times 1$	$= 1$
y	$= v_5$	$= 11.6521$	$\bar{v}_5$	$= \bar{y}$	$= 1$	

inputs $\dot{x}_i = 1$. With given values of x_i, a forward run would then compute derivatives of $\dot{y}_j = \frac{\partial y_j}{\partial x_i}, j = 1, \ldots, m$. Forward mode is ideal for functions $f : \mathbb{R} \to \mathbb{R}^m$, as all the required derivatives $\frac{\partial y_j}{\partial x}$ can be calculated with one forward pass. Conversely, in the other extreme of $f : \mathbb{R}^n \to \mathbb{R}$, forward mode would require n forward passes to compute all $\frac{\partial y}{\partial x_i}$. In general, for $f : \mathbb{R}^n \to \mathbb{R}^m$ where $n \gg m$, reverse AD is faster.

3.2. Reverse Mode

Like its familiar cousin backpropagation, reverse AD works by propagating derivatives backward from an output. It does this by supplementing each v_i with an adjoint $\bar{v}_i = \frac{\partial y_j}{\partial v_i}$ representing the sensitivity of output y_j to v_i. Derivatives are found in two stages: First, the original function is evaluated *forward*, computing v_i that will be subsequently needed. Second, derivatives are calculated in *reverse* by propagating $\bar{v}_i$ from the output to the inputs. In Table 2, the backward sweep of adjoints on the right-hand side starts with $\frac{\partial y}{\bar{v}_5} = \bar{y} = 1$ and we get both derivatives $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ in just one reverse sweep.

An advantage of reverse mode is that it is significantly less costly to evaluate than forward mode for functions with a large number of input variables—at least, in terms of operation count. In the case of $f : \mathbb{R}^n \to \mathbb{R}$, only one application of reverse mode would be sufficient to compute all partial derivatives $\frac{\partial y}{\partial x_i} = \bar{x}_i$, compared with the n sweeps that forward mode would need. In general, for an $f : \mathbb{R}^n \to \mathbb{R}^m$, if $time(f)$ is required for evaluating f, the time it takes to calculate the $m \times n$ Jacobian by forward AD is $O(n\, time(f))$, whereas the same can be done via reverse AD in $O(m\, time(f))$. That is to say, reverse mode AD performs better when $m \ll n$. On the other hand, Forward AD has only a constant factor overhead in space, while Reverse AD requires storage of intermediate results which increases its space complexity.

4. Derivatives and Machine Learning

Machine learning applications where computation of derivatives is necessary can include optimization, regression analysis, ANNs, support vector machines, clustering, and parame-

ter estimation. Let us examine some main uses of derivatives in machine learning and how these can benefit from the use of AD.

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, classical gradient descent has the goal of finding a (local) minimum $w^* = \arg\min_w f(w)$ via updates $\Delta w = -\eta \frac{df}{dw}$, where η is the step size. These methods make use of the fact that f decreases steepest if one goes in the direction of the negative gradient. For large n, reverse mode AD provides a highly efficient and exact method for gradient calculation, as we have outlined.

More sophisticated quasi-Newton methods, such as the BFGS algorithm and its variant L-BFGS, use both the gradient and the Hessian H of a function. In practice, the full Hessian is not computed but approximated using rank-one updates derived from gradient evaluations. AD can be used here for efficiently computing an exact Hessian-vector product Hv, via applying forward mode on a gradient found through reverse mode. Thus, Hv is computed with $O(n)$ complexity, even though H is a $n \times n$ matrix (Pearlmutter, 1994). Hessians arising in large-scale applications are typically sparse. This sparsity, along with symmetry, can be exploited by AD techniques such as elimination on computational graph of the Hessian (Dixon, 1991) or matrix coloring and compression (Gebremedhin et al., 2009).

Another approach for improving the asymptotic rate of convergence of gradient methods is to use gain adaptation methods such as stochastic meta-descent (SMD), where stochastic sampling is introduced to avoid local minima. An example using SMD with AD is given by Vishwanathan et al. (2006) on conditional random fields (CRF), for the probabilistic segmentation of data.

In ANNs, training is an optimization task with respect to the set of weights, which can in principle be attacked via any method including stochastic gradient descent or BFGS (Apostolopoulou et al., 2009). As we have pointed out, the highly successful backpropagation algorithm is a special case of reverse mode AD and there are instances in literature—albeit few—where ANNs are trained with explicit reference to AD, such as Eriksson et al. (1998) using AD for large-scale feed-forward networks, and Yang et al. (2008) where AD is used to train an ANN-based PID controller. Beyond backpropagation, the generality of AD opens up new possibilities. An example is given for continuous time recurrent neural networks (CTRNN) by Al Seyab and Cao (2008), where AD is used for training CTRNNs predicting dynamic behavior of nonlinear processes in real time. AD is used to calculate derivatives higher than second order, resulting in significantly reduced network training times as compared with other methods.

In computer vision, first and second order derivatives play an important role in tasks such as edge detection and sharpening (Russ, 2010). However, in most applications, these fundamental operations are applied on discrete functions of integer coordinates, approximating those derived on a hypothetical continuous spatial image function. As a consequence, derivatives are approximated using numerical differences. On the other hand, some computer vision tasks can be formulated as minimization of appropriate energy functionals. This minimization is usually accomplished via calculus of variations and the Euler-Lagrange equation, opening up the possibility of taking advantage of AD. In this area, the first study introducing AD to computer vision is given by Pock et al. (2007) which considers denoising, segmentation, and information recovery from stereoscopic image pairs and notes the benefit of AD in isolating sparsity patterns in large Jacobian and Hessian matrices. Grabner et al. (2008) use reverse AD for GPU-accelerated medical 2D/3D registration, a task concerning the alignment of data from different sources such as X-ray images or computed tomogra-

phy. A six-fold increase in speed (compared with numerical differentiation using center difference) is reported.

Nested applications of AD would facilitate compositional approaches to machine learning tasks, where one can, for example, perform gradient optimization on a system of many components that can in turn be internally using other derivatives or performing optimization (Siskind and Pearlmutter, 2008b; Radul et al., 2012). This capability is relevant to, e.g., hyperparameter optimization, where using gradient methods on model selection criteria has been proposed as an alternative to the established grid search and randomized search methods. Examples include the application to linear regression and time-series prediction (Bengio, 2000) and support vector machines (Chapelle et al., 2002).

It is important to note that AD is applicable to not only mathematical expressions in classical sense, but also algorithms of arbitrary structure, including those with control flow statements (Figure 1). Computations involving if-statements, loops, and procedure calls are in the end evaluated as straight-line traces of elementary operations—i.e. conditionals turned into actual paths taken, loops unrolled, and procedure calls inlined. In contrast, symbolic methods cannot be applied to such algorithms without significant manual effort.

A concerted effort to generalize AD to make it suitable for a wider range of machine learning tasks has been undertaken (Pearlmutter and Siskind, 2008). The resulting AD-enabled research prototype compilers generate very efficient code (Siskind and Pearlmutter, 2008a, also DVL https://github.com/axch/dysvunctional-language/), but these technologies are not yet available in production-quality systems.

5. Implementations

In practice, AD is used via feeding an existing algorithm into a tool, which augments it with the corresponding extra code to compute derivatives. This can be implemented through calls to a library; as a source transformation where a given code is automatically modified; or through operator overloading, which makes the process transparent to the user. Implementations exist for most programming languages[2] and a taxonomy of tools is given by Bischof et al. (2008).

6. Conclusions

The ubiquity of differentiation in machine learning renders AD a highly capable tool for the field. Needless to say, there are occasions where we are interested in obtaining more than just the numerical values for derivatives. Symbolic methods can be useful for analysis and gaining insight into the problem domain. However, for any non-trivial function of more than a handful of variables, analytic expressions for gradients or Hessians increase rapidly in complexity to render any interpretation unlikely.

Combining the expressive power of AD operators and functional programming would allow very concise implementations for a range of machine learning applications, which we intend to discuss in an upcoming article.

2. The website http://www.autodiff.org/ maintains a list of implementations.

Acknowledgments

This work was supported in part by Science Foundation Ireland grant 09/IN.1/I2637.

References

R. K. Al Seyab and Y. Cao. Nonlinear system identification for predictive control using continuous time recurrent neural networks and automatic differentiation. *Journal of Process Control*, 18(6):568–581, 2008.

M. S. Apostolopoulou, D. G. Sotiropoulos, I. E. Livieris, and P. Pintelas. A memoryless BFGS neural network training algorithm. In *INDIN 2009*, pages 216–221, 2009.

Y. Bengio. Gradient-based optimization of hyper-parameters. *Neural Computation*, 12(8), 2000.

C. H. Bischof, P. D. Hovland, and B. Norris. On the implementation of automatic differentiation tools. *Higher-Order and Symbolic Computation*, 21(3):311–31, 2008.

O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46:131–159, 2002.

L. C. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In *Automatic Differentiation of Algorithms*, pages 114–125. SIAM, 1991.

J. Eriksson, M. Gulliksson, P. Lindström, and P. Wedin. Regularization tools for training large feed-forward neural networks using automatic differentiation. *Optimization Methods and Software*, 10(1):49–69, 1998.

A. Gebremedhin, A. Pothen, A. Tarafdar, and A. Walther. Efficient computation of sparse Hessians using coloring and automatic differentiation. *INFORMS Journal on Computing*, 21(2):209–23, 2009.

M. Grabner, T. Pock, T. Gross, and B. Kainz. Automatic differentiation for GPU-accelerated 2D/3D registration. In *Advances in Automatic Differentiation*, volume 64, pages 259–269. Springer, 2008.

A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, 2008.

R. Neal. MCMC for using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, pages 113–62, 2011.

B. A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6:147–60, 1994.

B. A. Pearlmutter and J. M. Siskind. Using programming language theory to make AD sound and efficient. In *AD 2008, Bonn, Germany*, pages 79–90, 2008.

T. Pock, M. Pock, and H. Bischof. Algorithmic differentiation: Application to variational problems in computer vision. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 29(7):1180–1193, 2007.

A. Radul, B. A. Pearlmutter, and J. M. Siskind. AD in Fortran, Part 1: Design. Technical Report arXiv:1203.1448, 2012.

J. C. Russ. *The Image Processing Handbook*. CRC press, 2010.

J. M. Siskind and B. A. Pearlmutter. Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient Fortran-like code. Technical Report TR-ECE-08-01, ECE, Purdue Univ., 2008a.

J. M. Siskind and B. A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–76, 2008b.

S. Sra, S. Nowozin, and S. J. Wright. *Optimization for Machine Learning*. MIT Press, 2011.

S. V. N. Vishwanathan, N. N. Schraudolph, M. W. Schmidt, and K. P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pages 969–76, 2006.

A. Walther. Automatic differentiation of explicit Runge-Kutta methods for optimal control. *Computational Optimization and Applications*, 36(1):83–108, 2007.

W. Yang, Y. Zhao, L. Yan, and X. Chen. Application of PID controller based on BP neural network using automatic differentiation method. In *Advances in Neural Networks*, volume 5264, pages 702–711. 2008.