# Automatic differentiation in machine learning: A survey

**Article** *in* Journal of Machine Learning Research · April 2018

Source: arXiv

CITATIONS
2,655

READS
3,446

**4 authors**, including:

Barak A. Pearlmutter
National University of Ireland, Maynooth
**182** PUBLICATIONS **11,087** CITATIONS
SEE PROFILE

Jeffrey Siskind
Purdue University
**116** PUBLICATIONS **6,446** CITATIONS
SEE PROFILE

**Mach Learn manuscript No.**
(will be inserted by the editor)

# Automatic differentiation in machine learning: a survey

**Atılım Güneş Baydin · Barak A. Pearlmutter · Alexey Andreyevich Radul**



Received: date / Accepted: date

**Abstract** Derivatives, mostly in the form of gradients and Hessians, are ubiquitous in machine learning. Automatic differentiation (AD) is a technique for calculating derivatives efficiently and accurately, established in fields such as computational fluid dynamics, nuclear engineering, and atmospheric sciences. Despite its advantages and use in other fields, machine learning practitioners have been little influenced by AD and make scant use of available tools. We survey the intersection of AD and machine learning, cover applications where AD has the potential to make a big impact, and report on the recent developments in the adoption this technique. We also aim to dispel some misconceptions that we think have impeded the widespread awareness of AD within the machine learning community.

**Keywords** Automatic differentiation · Optimization · Gradient methods · Backpropagation

## 1 Introduction

The computation of derivatives in computer models is addressed by four main methods: 1. manually working out derivatives and coding the result; 2. numerical differentiation (using finite difference approximations); 3. symbolic differentiation (using expression-manipulation in of software such as Maxima, Mathematica, and Maple); and 4. automatic differentiation.

A. G. Baydin (✉) · B. A. Pearlmutter
Hamilton Institute & Department of Computer Science
National University of Ireland Maynooth, Maynooth, Co. Kildare, Ireland
E-mail: atilimgunes.baydin@nuim.ie; barak@cs.nuim.ie

A. A. Radul
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, MA 02139, United States
E-mail: axofch@gmail.com

Classically, many methods in machine learning require the evaluation of derivatives and most of the traditional learning algorithms rely on the computation of gradients and Hessians of an objective function (Sra et al, 2011). Examples include the training of artificial neural networks (Widrow and Lehr, 1990), conditional random fields (Vishwanathan et al, 2006), natural language processing (Finkel et al, 2008), and computer vision (Parker, 2010).

When introducing new models, machine learning researchers spend considerable effort on the manual derivation of analytical derivatives and subsequently plug these into standard optimization procedures such as L-BFGS (Zhu et al, 1997) or stochastic gradient descent (Bottou, 1998).

Manual differentiation is evidently time consuming and prone to error. Of the other alternatives, numerical differentiation is simple to implement, but susceptible to round-off and truncation errors that make it inherently unstable (Jerrell, 1997). Symbolic differentiation addresses the weaknesses of both the manual and numerical methods, but often results in complex and cryptic expressions plagued with the problem of "expression swell" (Corliss, 1988). Furthermore, manual and symbolic methods require the model to be expressed as a closed-form mathematical formula, ruling out algorithmic control flow and severely limiting expressivity.

We are concerned with the powerful fourth technique, automatic differentiation[1] (AD), which works by systematically applying the chain rule of differential calculus at the elementary operator level. AD allows the accurate evaluation of derivatives in machine precision, with only a small constant factor of overhead and ideal asymptotic efficiency. In contrast with the effort involved in arranging code into closed-form expressions for symbolic differentiation, AD can usually be applied to existing code with minimal change. Because of its generality, AD is an already established tool in applications including real-parameter optimization (Walther, 2007), sensitivity analysis (Carmichael and Sandu, 1997), physical modeling (Ekström et al, 2010), and probabilistic inference (Neal, 2011).

Despite its widespread use in other fields, AD has been underused, if not unknown, by the machine learning community.

As it happens, AD and machine learning practice are conceptually very closely interconnected: consider the backpropagation method for training neural networks, which has a colorful history of being rediscovered several times by independent researchers (Widrow and Lehr, 1990). It has been one of the most studied and used training algorithms since the day it became popular mainly through the work of Rumelhart et al (1986). In simplest terms, backpropagation models learning as gradient descent in neural network weight space, looking for the minimum of an error function (Gori and Maggini, 1996). This is accomplished by the backwards propagation of the error values at the output (Fig. 1) utilizing the chain rule to compute the gradient of the error at each point. The resulting algorithm is essentially equivalent to transforming the network evaluation function through the so-called *reverse mode AD*, which, as we will see, actually generalizes the backpropagation idea. Thus, a modest un-

[1] Also called *algorithmic differentiation*, and less frequently *computational differentiation*.

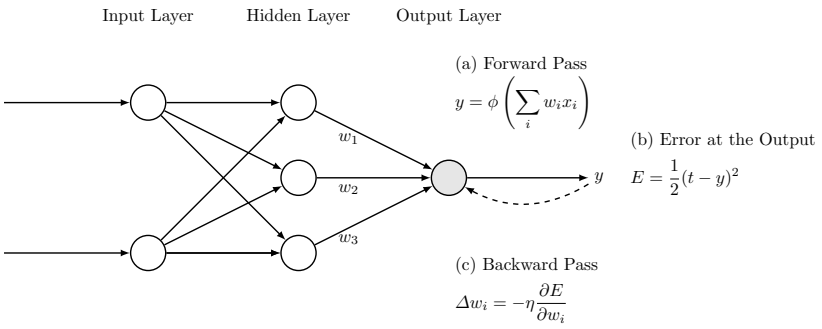


**Fig. 1** Overview of backpropagation. (a) Training pattern is fed forward, generating corresponding output. (b) Error between actual and desired output is computed. (c) The error propagates back, through updates where a ratio of the gradient ($\frac{\partial E}{\partial w_i}$) is subtracted from each weight. $x_i$, $w_i$, $\phi$ are the inputs, input weights, and activation function of a neuron. Error $E$ is computed from output $y$ and desired output $t$. $\eta$ is the learning rate.

derstanding of the mathematics underlying backpropagation already provides sufficient background for grasping the AD technique.

Here we review the AD technique from a machine learning perspective, covering its origins, potential applications in machine learning, and methods of its implementation. It is our hope that the review will be a concise and accessible introduction to the technique for machine learning practitioners. Along the way, we also aim to dispel some misconceptions that we believe have impeded wider appraisal of AD.

The article is structured as follows: in Sect. 2 we start by emphasizing how AD actually differs from numerical and symbolic differentiation; Sect. 3 gives an introduction to the technique and its forward and reverse modes of operation; Sect. 4 discusses the role of derivatives in machine learning and examines cases where AD has the potential to have an impact; Sect. 5 covers the implementation approaches and available AD tools; and Sect. 6 offers our conclusions.

## 2 What AD is not

Without proper introduction, the term "automatic differentiation" has undertones suggesting that it is either a type of symbolic or numerical differentiation. This can be intensified by the dichotomy that the final results from AD are indeed numerical values, while the steps in its computation do depend on algebraic manipulation, giving AD a two-sided nature that is partly symbolic and partly numerical (Griewank, 2003).

Let us start by stressing how AD is different from, and in some aspects superior to, these two commonly encountered techniques of differentiation.

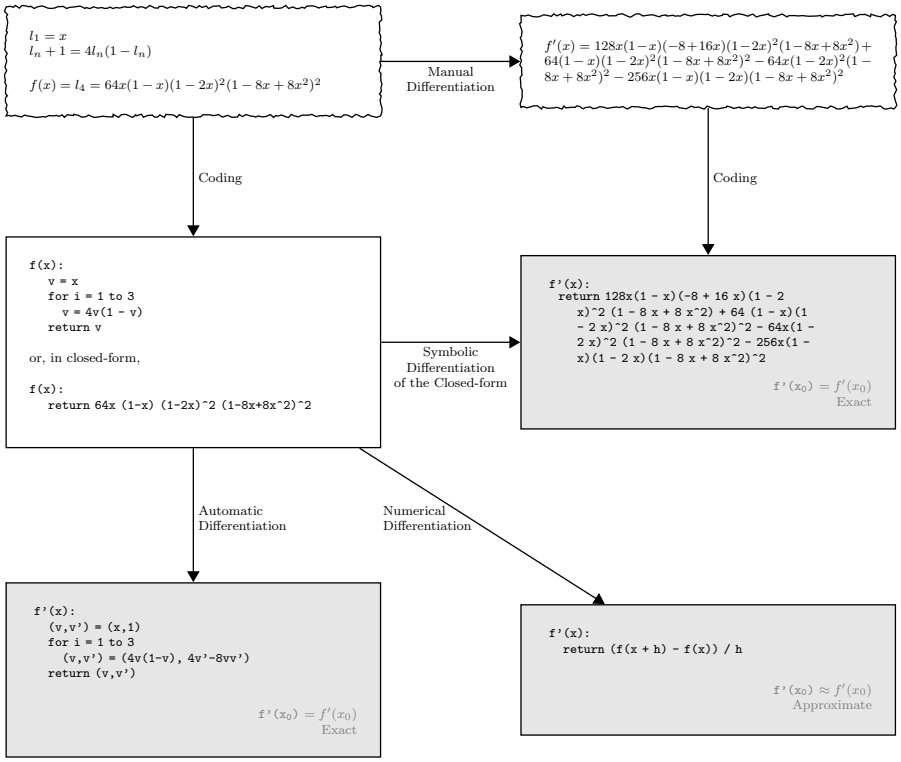


**Fig. 2** The range of approaches for differentiating mathematical expressions and computer code. Symbolic differentiation (center right) gives exact results but suffers from unbounded expression swell; numeric differentiation (lower right) has problems of accuracy due to round-off and truncation errors; automatic differentiation (lower left) is as accurate as symbolic differentiation with only a constant factor of overhead.

## 2.1 AD is not numerical differentiation

Numerical differentiation is the finite difference approximation of derivatives using values of the original function evaluated at some sample points (Burden and Faires, 2001) (Fig. 2). In its simplest form, it is based on the standard definition of a derivative. For example, for a function of many variables $f : \mathbb{R}^n \to \mathbb{R}$, we can approximate the gradient $\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$ using

$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h} , \tag{1}$$

where $\mathbf{e}_i$ is the $i$-th unit vector and $0 < h \ll 1$ is a step size. This has the advantage of being uncomplicated to implement, but the disadvantages of costing $O(n)$ evaluations of $f$ for a gradient in $n$ dimensions and requiring careful consideration in selecting the step size $h$.

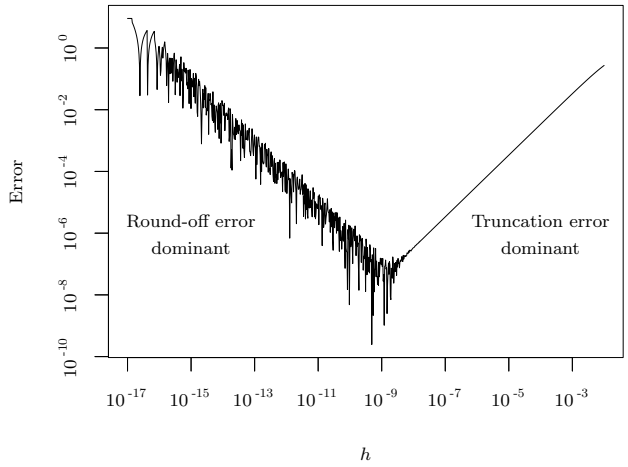


**Fig. 3** Error in the forward difference approximation (Eq. 1) as a function of step size $h$, for the derivative of $f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$. Computed using $E(h, x_0) = \left| \frac{f(x_0+h)-f(x_0)}{h} - \frac{d}{dx} f(x)\big|_{x_0} \right|$ at $x_0 = 0.2$ .

Numerical approximations of derivatives are inherently ill-conditioned and unstable[2], with the exception of complex variable methods that are applicable to a limited set of holomorphic functions (Fornberg, 1981). This is due to the introduction of truncation[3] and round-off[4] errors, inflicted by the limited precision of computations and the chosen value of the step size $h$. Truncation error tends to zero as $h \to 0$. However, as $h$ is decreased, round-off error increases and becomes dominant (Fig. 3).

Improved techniques have been conceived to mitigate this shortcoming of numerical differentiation, such as using a center difference approximation

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h} + O(h^2) \,, \tag{2}$$

where the first-order errors cancel and effectively move the truncation error from first-order to second-order[5] in $h$. For the one-dimensional case, it is just as costly to compute the forward difference (Eq. 1) and the center difference (Eq. 2), requiring only two evaluations of $f$. However, with increasing

[2] Using the limit definition of the derivative for finite difference approximation commits both cardinal sins of numerical analysis: *"thou shalt not add small numbers to big numbers"*, and *"thou shalt not subtract numbers which are approximately equal"*.

[3] Truncation error is the error of approximation, or inaccuracy, one gets from $h$ not actually being zero. It is proportional to a power of $h$.

[4] Round-off error is the inaccuracy one gets from valuable low-order bits of the final answer having to compete for machine-word space with high-order bits of $f(\mathbf{x} + h\mathbf{e}_i)$ and $f(\mathbf{x})$ (Eq. 1), which the computer has to store just until they cancel in the subtraction at the end. Round-off error is inversely proportional to a power of $h$.

[5] This does not avoid either of the cardinal sins, and is still highly inaccurate due to truncation.

dimensionality, a trade-off between accuracy and performance is faced, where computing a Jacobian matrix of an $f : \mathbb{R}^n \to \mathbb{R}^m$ requires $2mn$ evaluations.

Other techniques for improving numerical differentiation, including higher-order finite differences, Richardson extrapolation to the limit (Brezinski and Zaglia, 1991), and differential quadrature methods using weighted sums (Bert and Malik, 1996), also increase rapidly in programming complexity, do not completely eliminate approximation errors, and remain highly susceptible to floating point truncation.

### 2.2 AD is not symbolic differentiation

Symbolic differentiation is the automatic manipulation of expressions for obtaining derivatives (Grabmeier et al, 2003) (Fig. 2). It is carried out by computer algebra packages that implement differentiation rules such as

$$\begin{aligned} \frac{d}{dx}\left(f(x)+g(x)\right) &= \frac{d}{dx}f(x)+\frac{d}{dx}g(x) \quad or \\ \frac{d}{dx}\left(f(x)\,g(x)\right) &= \left(\frac{d}{dx}f(x)\right)g(x)+f(x)\left(\frac{d}{dx}g(x)\right) . \end{aligned} \tag{3}$$

When formulae are represented as data structures, symbolically differentiating an expression tree is a perfectly mechanistic process, already considered subject to mechanical automation at the very inception of calculus (Leibniz, 1685). This is realized in modern computer algebra systems such as Mathematica, Maple, and Maxima.

In optimization, symbolic differentiation can give valuable insight into the structure of the problem domain and, in some cases, produce analytical solutions of extrema (e.g. $\frac{d}{dx}f(x) = 0$) that can eliminate the calculation of derivatives altogether. On the other hand, symbolic derivatives do not lend themselves to efficient run-time calculation of derivative values, as they can be exponentially larger than the expression whose derivative they represent.

Consider a function $h(x) = f(x)g(x)$ and the multiplication rule in Eq. 3. Since $h$ is a product, $h(x)$ and $\frac{d}{dx}h(x)$ have some common components (namely $f(x)$ and $g(x)$). Notice also that on the right hand side, $f(x)$ and $\frac{d}{dx}f(x)$ appear separately. If we just proceed to symbolically differentiate $f(x)$ and plug its derivative into the appropriate place, we will have nested duplications of any computation that appears in common between $f(x)$ and $\frac{d}{dx}f(x)$. Hence, careless symbolic differentiation can easily produce exponentially large symbolic expressions which take correspondingly long to evaluate. This problem is known as *expression swell* (Table 1).

When we are concerned with the accurate computation of derivative values and not so much with their actual symbolic form, it is in principle possible to simplify computations by storing values of intermediate subexpressions in memory. Moreover, for further efficiency, we can interleave as much as possible the differentiating and simplifying steps.

**Table 1** Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$ and the corresponding derivatives of $l_n$ with respect to $x$, illustrating expression swell.

| $n$ | $l_n$ | $\frac{d}{dx} l_n$ | $\frac{d}{dx} l_n$ (Optimized) |
|---|---|---|---|
| 1 | $x$ | 1 | 1 |
| 2 | $4x(1-x)$ | $4(1-x) - 4x$ | $4 - 8x$ |
| 3 | $16x(1-x)(1-2x)^2$ | $16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$ | $16(1 - 10x + 24x^2 - 16x^3)$ |
| 4 | $64x(1-x)(1-2x)^2(1-8x+8x^2)^2$ | $128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2) + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$ | $64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$ |

This "interleaving" idea forms the basis of AD and provides an account of its simplest form: *apply symbolic differentiation in elementary operations level and keep intermediate numerical results, in lockstep with the evaluation of the main function.*

## 3 Preliminaries

In its most basic description, AD relies on the fact that all computations are ultimately compositions of a finite set of elementary operations for which derivatives are known (Verma, 2000). Combining the derivatives of constituent operations through the chain rule gives the derivative of the overall composition. Usually, these elementary operations include the binary operations $+$ and $\times$, the unary sign switch $-$, the reciprocal, and the standard univariate functions such as exp, sin and the like.

On the left hand side of Table 2 we see the representation of the computation $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ as an *evaluation trace* of elementary operations—also called a Wengert list (Wengert, 1964). We adopt the "three-part notation" used by Griewank and Walther (2008), where a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is constructed using intermediate variables $v_i$ such that

- variables $v_{i-n} = x_i,\ i = 1, \ldots, n$ are the input variables,
- variables $v_i\ i = 1, \ldots, l$ are the working variables, and
- variables $y_{m-i} = v_{l-i},\ i = m-1, \ldots, 0$ are the output variables.

A given trace of elementary operations can be also represented using a computational graph (Bauer, 1974), as shown in Fig. 4. Such graphs are useful in visualizing dependency relations between intermediate variables.

Evaluation traces form the basis of the AD technique. An important point to note here is that any numeric code will eventually be run—or evaluated—as a trace, with particular input values and the resulting output. Thus, AD

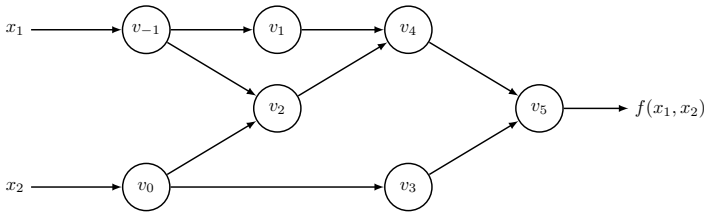


**Fig. 4** Computational graph of the example $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. See Tables 2 and 3 for the definitions of the intermediate variables $v_{-1} \ldots v_5$ .

can differentiate not only mathematical expressions in the classical sense, but also algorithms making use of control flow statements, loops, and procedure calls. This gives AD an important advantage over symbolic differentiation which can only be applied after arranging code into closed-form mathematical expressions.

### 3.1 Forward mode

Forward mode[6] is conceptually the most simple type of AD.

Consider the evaluation trace of the function $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ given on the left hand side of Table 2 and in graph form in Fig. 4. For computing the derivative of $f$ with respect to $x_1$, we start by associating with each intermediate variable $v_i$ a derivative

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} .$$

Applying the chain rule to each elementary operation in the forward evaluation trace, we generate the corresponding derivative trace, given on the right hand side of Table 2. Evaluating variables $v_i$ one by one together with their corresponding $\dot{v}_i$ values gives us the required derivative in the final variable $\dot{v}_5 = \frac{\partial y}{\partial x_1}$.

This generalizes naturally to computing the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with $n$ independent variables $x_i$ and $m$ dependent variables $y_j$. In this case, each forward pass of AD is initialized by setting only one of the variables $\dot{x}_i = 1$ (in other words, setting $\dot{\mathbf{x}} = \mathbf{e}_i$, where $\mathbf{e}_i$ is the $i$-th unit vector). A run of the code with specific input values $\mathbf{x} = \mathbf{a}$ would then compute

$$\dot{y}_j = \left.\frac{\partial y_j}{\partial x_i}\right|_{\mathbf{x}=\mathbf{a}} , \; j = 1, \ldots, m ,$$

[6] Also called *tangent linear* mode.

**Table 2** Forward mode 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 $\frac{\partial y}{\partial x_1}$. The original forward run on the left is augmented by the forward AD operations on the right, where each line supplements the original on its left.

| Forward Evaluation Trace | | | Forward Derivative Trace | | |
|---|---|---|---|---|---|
| $v_{-1}$ | $= x_1$ | $= 2$ | $\dot{v}_{-1}$ | $= \dot{x}_1$ | $= 1$ |
| $v_0$ | $= x_2$ | $= 5$ | $\dot{v}_0$ | $= \dot{x}_2$ | $= 0$ |
| $v_1$ | $= \ln v_{-1}$ | $= \ln 2$ | $\dot{v}_1$ | $= \dot{v}_{-1}/v_{-1}$ | $= 1/2$ |
| $v_2$ | $= v_{-1} \times v_0$ | $= 2 \times 5$ | $\dot{v}_2$ | $= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$ | $= 1 \times 5 + 0 \times 2$ |
| $v_3$ | $= \sin v_0$ | $= \sin 5$ | $\dot{v}_3$ | $= \dot{v}_0 \times \cos v_0$ | $= 0 \times \cos 5$ |
| $v_4$ | $= v_1 + v_2$ | $= 0.693 + 10$ | $\dot{v}_4$ | $= \dot{v}_1 + \dot{v}_2$ | $= 0.5 + 5$ |
| $v_5$ | $= v_4 - v_3$ | $= 10.693 + 0.959$ | $\dot{v}_5$ | $= \dot{v}_4 - \dot{v}_3$ | $= 5.5 - 0$ |
| $y$ | $= v_5$ | $= 11.652$ | $\dot{y}$ | $= \dot{v}_5$ | $= 5.5$ |

giving us one column of the Jacobian matrix

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \Bigg|_{\mathbf{x}\,=\,\mathbf{a}}$$

evaluated at point $\mathbf{a}$. Thus, the full Jacobian can be computed in $n$ evaluations.

Furthermore, forward mode AD provides a very efficient and matrix-free way of computing Jacobian-vector products

$$\mathbf{J}_f\,\mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}, \tag{4}$$

simply by initializing with $\dot{\mathbf{x}} = \mathbf{r}$. Thus, we can compute the Jacobian-vector product in just one forward pass. Similarly, for a function $f : \mathbb{R}^n \to \mathbb{R}$, we can obtain the directional derivative along a given vector $\mathbf{r}$ as a linear combination of the partial derivatives

$$\nabla f \cdot \mathbf{r}$$

via starting the AD computation with the values $\dot{\mathbf{x}} = \mathbf{r}$.

Forward mode AD is efficient and straightforward for functions $f : \mathbb{R} \to \mathbb{R}^m$, as all the derivatives $\frac{\partial y_i}{\partial x}$ can be computed with just one forward pass. Conversely, in the other extreme of $f : \mathbb{R}^n \to \mathbb{R}$, forward mode AD would require $n$ evaluations to compute the gradient

$$\nabla f = \left( \frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n} \right).$$

In general, for cases $f : \mathbb{R}^n \to \mathbb{R}^m$ where $n \gg m$, a superior technique, the reverse mode AD is preferred.

### *3.1.1 Dual numbers*

A common way to handle the paired operations in forward mode AD (represented by the left and right hand sides in Table 2) is through the use of dual numbers[7], which are defined as formal truncated Taylor series of the form

$$x + x'\epsilon \, .$$

Defining arithmetic on dual numbers by $\epsilon^2 = 0$ and by interpreting any non-dual number $y$ as $y + 0\epsilon$, we get entities such as

$$\begin{aligned}(x + x'\epsilon) + (y + y'\epsilon) &= (x + y) + (x' + y')\epsilon \, , \\ (x + x'\epsilon)(y + y'\epsilon) &= (xy) + (xy' + x'y)\epsilon \, ,\end{aligned}$$

in which the coefficients of $\epsilon$ conveniently mirror symbolic differentiation rules (e.g. Eq. 3). We can utilize this by setting up a regime where

$$f(x + x'\epsilon) = f(x) + f'(x)x'\epsilon \tag{5}$$

and using dual numbers as data structures for carrying the derivative together with the undifferentiated value. The chain rule works as expected on this representation. For example, two applications of Eq. 5 give

$$\begin{aligned}f(g(x + x'\epsilon)) &= f(g(x) + g'(x)x'\epsilon) \\ &= f(g(x)) + f'(g(x))g'(x)x'\epsilon \, ,\end{aligned}$$

where the coefficient of $\epsilon$ on the right hand side is exactly the derivative of the composition of $f$ and $g$. This means that since we implement elementary operations to respect the invariant Eq. 5, all compositions of them will also do so. This, in turn, means that we can extract the derivative of a function of interest by evaluating it in this nonstandard way on an initial input with a coefficient 1 for $\epsilon$:

$$\left.\frac{df(x)}{dx}\right|_x = \text{epsilon-coefficient}(\text{dual-version}(f)(x + 1\epsilon)) \, .$$

This also extends to arbitrary program constructs, since dual numbers, as data types, can be contained in any data structure. As long as no arithmetic is done on the dual number, it will just remain a dual number; and if it is taken out of the data structure and operated on again, then the differentiation will continue. If we look at it from the point of view of the dual number, AD amounts to partially evaluating $f$ with respect to that input to derive a symbolic expression, symbolically differentiating that expression, and collapsing the result (in a particular way) to make sure it is not too big; all interleaved to prevent intermediate expression swells.

In practice, a function $f$ coded in a programming language of choice would be fed into an AD tool, which would then augment it with corresponding extra code to handle the dual operations, so that the function and its derivative are

[7] First introduced by Clifford (1873), with important uses in linear algebra and physics.

simultaneously computed. This can be implemented through calls to a specific library; in the form of source transformation where a given source code will be automatically modified; or through operator overloading, making the process transparent to the user. We cover the implementation techniques in Sect. 5.

### 3.2 Reverse mode

Like its famous cousin backpropagation, reverse mode[8] AD works by propagating derivatives backward from a given output. It does this by supplementing each intermediate variable $v_i$ with an adjoint

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i} ,$$

which represents the sensitivity of a considered output $y_j$ with respect to changes in $v_i$.

Derivatives are then computed using what is essentially a two stage process. In the first stage, the original function code is run *forward*, populating intermediate variables $v_i$ and keeping track of the dependencies in the computational graph. In the second stage, derivatives are calculated by propagating adjoints $\bar{v}_i$ in *reverse*, from the outputs to the inputs.

Returning to the example $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$, in Table 3 we see the adjoint statements on the right hand side, corresponding to each original elementary operation on the left. In simple terms, we are interested in computing the contribution $\bar{v}_i = \frac{\partial y}{\partial v_i}$ of the change in each variable $v_i$ on the change in the output $y$. Taking the variable $v_0$ as an example, we see (Fig. 4) that the only ways it can affect $y$ are through $v_2$ and $v_3$, so its contribution to the change in $y$ is given by

$$\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0} , \ or$$
$$\bar{v}_0 = \bar{v}_2 \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \frac{\partial v_3}{\partial v_0} .$$

In Table 3, this contribution is computed in two incremental steps

$$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0} \quad and$$
$$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} ,$$

grouped with the line in the original trace from which it originates.

After the forward sweep on the left hand side, we run the reverse sweep of the adjoints on the right hand side, starting with $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$. In the end we get the derivatives $\frac{\partial y}{\partial x_1} = \bar{x}_1$ and $\frac{\partial y}{\partial x_2} = \bar{x}_2$ in just one reverse sweep.

[8] Also called *adjoint* or *cotangent linear* mode.

**Table 3** Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ at $(x_1, x_2) = (2, 5)$. After running the original forward run on the left, the augmented AD operations on the right are run in reverse (cf. Fig. 1). Both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse sweep, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

| Forward Evaluation Trace | | | Reverse Adjoint Trace | | | |
|---|---|---|---|---|---|---|
| $v_{-1}$ | $= x_1$ | $= 2$ | $\bar{\boldsymbol{x}}_{\mathbf{1}}$ | $= \bar{\boldsymbol{v}}_{\boldsymbol{-1}}$ | | $\mathbf{= 5.5}$ |
| $v_0$ | $= x_2$ | $= 5$ | $\bar{\boldsymbol{x}}_{\mathbf{2}}$ | $= \bar{\boldsymbol{v}}_{\mathbf{0}}$ | | $\mathbf{= 1.716}$ |
| $v_1$ | $= \ln v_{-1}$ | $= \ln 2$ | $\bar{v}_{-1}$ | $= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$ | $= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$ | $= 5.5$ |
| $v_2$ | $= v_{-1} \times v_0$ | $= 2 \times 5$ | $\bar{v}_0$ | $= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$ | $= \bar{v}_0 + \bar{v}_2 \times v_{-1}$ | $= 1.716$ |
| $v_3$ | $= \sin v_0$ | $= \sin 5$ | $\bar{v}_{-1}$ | $= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$ | $= \bar{v}_2 \times v_0$ | $= 5$ |
| $v_4$ | $= v_1 + v_2$ | $= 0.693 + 10$ | $\bar{v}_0$ | $= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$ | $= \bar{v}_3 \times \cos v_0$ | $= -0.284$ |
| $v_5$ | $= v_4 - v_3$ | $= 10.693 + 0.959$ | $\bar{v}_2$ | $= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$ | $= \bar{v}_4 \times 1$ | $= 1$ |
| | | | $\bar{v}_1$ | $= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$ | $= \bar{v}_4 \times 1$ | $= 1$ |
| | | | $\bar{v}_3$ | $= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$ | $= \bar{v}_5 \times (-1)$ | $= -1$ |
| | | | $\bar{v}_4$ | $= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$ | $= \bar{v}_5 \times 1$ | $= 1$ |
| $y$ | $= v_5$ | $= 11.652$ | $\bar{v}_5$ | $= \bar{y}$ | $= 1$ | |

Compared with the straightforward simplicity of the forward mode, reverse mode AD can, at first, appear somewhat "mysterious" (Dennis and Schnabel, 1996). Griewank and Walther (2008) argue that this is in part because of the common acquaintance with the chain rule as a mechanical procedure propagating derivatives forward.

An important advantage of the reverse mode is that it is significantly less costly to evaluate than the forward mode for functions with a large number of input variables. In the extreme case of $f : \mathbb{R}^n \to \mathbb{R}$, only one application of the reverse mode is sufficient to compute the full gradient $\nabla f = \left( \frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n} \right)$, compared with the $n$ sweeps of the forward mode needed for the same.

In general, for a function $f : \mathbb{R}^n \to \mathbb{R}^m$, if we denote the time it takes to evaluate the original function by $\text{time}(f)$, the time it takes to calculate the the $m \times n$ Jacobian by the forward mode is $O(n\,\text{time}(f))$, whereas the same computation can be done via reverse mode in $O(m\,\text{time}(f))$. That is to say, reverse mode AD performs better when $m \ll n$.

Similar to the matrix-free computation of Jacobian-vector products with the forward mode (Eq. 4), the reverse mode can be used for computing the transposed Jacobian-vector product

$$\mathbf{J}_f^\intercal \mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_m \end{bmatrix},$$

by initializing the reverse stage with $\bar{\mathbf{y}} = \mathbf{r}$.

The advantages of reverse mode AD, however, come with the cost of increased storage requirements growing in proportion to the number of oper-

ations in the evaluated function. It is an active area of research to improve storage requirements in implementations, by methods such as checkpointing strategies and data-flow analysis (Dauvergne and Hascoët, 2006).

### 3.3 Origins of AD and backpropagation

Ideas underlying the AD technique date back to the 1950s (Nolan, 1953; Beda et al, 1959). Forward mode AD as a general method for evaluating partial derivatives was essentially discovered by Wengert (1964). It was followed by a period of relatively low activity, until interest in the field was revived by the 1980s mostly through the work of Griewank (1989), also supported by improvements in modern programming languages and the feasibility of an efficient reverse mode AD[9].

Reverse mode AD and backpropagation have an intertwined history.

The essence of reverse mode AD, cast in a continuous-time formalism, is the Pontryagin Maximum principle (Rozonoer and Pontryagin, 1959; Goltyanskii et al, 1960). This method was understood in the control theory community (Bryson, 1962; Bryson and Ho, 1969) and cast in more formal terms with discrete-time variables topologically sorted in terms of dependency by Bryson's PhD student Werbos (1974). Speelpenning (1980) discovered reverse mode AD and gave the first implementation that was actually automatic, in the sense of accepting a specification of a computational process written in a general-purpose programming language and automatically performing the reverse mode transformation.

Incidentally, Hecht-Nielsen (1989) cites the work of Bryson and Ho (1969) and Werbos (1974) as the two earliest known instances of backpropagation. Within the machine learning community, the method has been reinvented several times, such as by Parker (1985), until it was eventually brought to fame by Rumelhart et al (1986) of the Parallel Distributed Processing (PDP) group. The PDP group became aware of Parker's work only after their own discovery, and similarly, Werbos' work was not appreciated until it was found by Parker.

This tells us an interesting story of two highly interconnected research communities that have somehow also managed to stay detached during this foundational period.

## 4 Derivatives and machine learning

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

Classically, the main tasks in machine learning where the computation of derivatives is relevant have included optimization, various models of regression analysis (Draper and Smith, 1998), neural networks (Widrow and Lehr, 1990), clustering, computer vision, and parameter estimation.

[9] For a thorough review of the development of AD, we advise readers to refer to Rall (2006). Also see Griewank (2012) for an investigation of the origins of the reverse mode.

**Table 4** Evaluation times of the Helmholtz free energy function and its gradient (Fig. 5). The times are given relative to that of the original function with $n = 1$ and with $n$ corresponding to each column. (For instance, reverse mode AD with $n = 43$ takes approximately twice the time to evaluate relative to the original function with $n = 43$.) Times are measured by averaging a thousand runs on a Windows 8.1 machine with Intel Core i7-4785T 2.20 GHz CPU and 16 GB RAM, using the DiffSharp AD library v0.5.7. The original function with $n = 1$ was evaluated in 0.0023 ms.

| | $n$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 8 | 15 | 22 | 29 | 36 | 43 | 50 |
| $f$, original | | | | | | | | |
| Rel. $n = 1$ | 1 | 5.12 | 14.51 | 29.11 | 52.58 | 84.00 | 127.33 | 174.44 |
| $\nabla f$, num. diff. | | | | | | | | |
| Rel. $n = 1$ | 1.08 | 35.55 | 176.79 | 499.43 | 1045.29 | 1986.70 | 3269.36 | 4995.96 |
| Rel. col. | 1.08 | 6.93 | 12.17 | 17.15 | 19.87 | 23.64 | 25.67 | 28.63 |
| $\nabla f$, forward AD | | | | | | | | |
| Rel. $n = 1$ | 1.34 | 13.69 | 51.54 | 132.33 | 251.32 | 469.84 | 815.55 | 1342.07 |
| Rel. col. | 1.34 | 2.66 | 3.55 | 4.54 | 4.77 | 5.59 | 6.40 | 7.69 |
| $\nabla f$, reverse AD | | | | | | | | |
| Rel. $n = 1$ | 1.52 | 11.12 | 31.37 | 67.27 | 113.99 | 174.62 | 254.15 | 342.33 |
| Rel. col. | 1.52 | 2.16 | 2.16 | 2.31 | 2.16 | 2.07 | 1.99 | 1.96 |

### 4.1 Gradient methods

Given an objective function $f : \mathbb{R}^n \to \mathbb{R}$, classical gradient descent has the goal of finding (local) minima $\mathbf{w}^* = \arg\min_{\mathbf{w}} f(\mathbf{w})$ via updates $\Delta\mathbf{w} = -\eta\nabla f$, where $\eta$ is a step size. Gradient methods make use of the fact that $f$ decreases steepest if one goes in the direction of the negative gradient. Naive gradient descent comes with asymptotic rate of convergence, where the method increasingly "zigzags" towards the minimum in a slowing down fashion. Convergence rate is usually improved by adaptive step size techniques that adjust the step size $\eta$ on every iteration (Snyman, 2005).

As we have seen, for large $n$, reverse mode AD provides a highly efficient method for computing gradients[10]. In Fig. 5 and Table 4, we demonstrate how gradient methods can benefit from AD, looking at the example of Helmholtz free energy function that has been used in AD literature (Griewank, 1989; Griewank and Walther, 2008) for benchmarking gradient calculations.

Second-order methods based on Newton's method make use of both the gradient $\nabla f$ and the Hessian $\mathbf{H}_f$, working via updates $\Delta\mathbf{w} = -\eta\,\mathbf{H}_f^{-1}\nabla f$. Newton's method converges in fewer iterations, but this comes with the cost of computing $\mathbf{H}_f$ in each step (Press et al, 2007). Due to its computational cost, the Hessian is usually replaced by a numerical approximation using updates from gradient evaluations, giving rise to quasi-Newton methods. A highly pop-

[10] See http://gbaydin.github.io/DiffSharp/examples-gradientdescent.html for an example of AD-based gradient descent using the DiffSharp library.

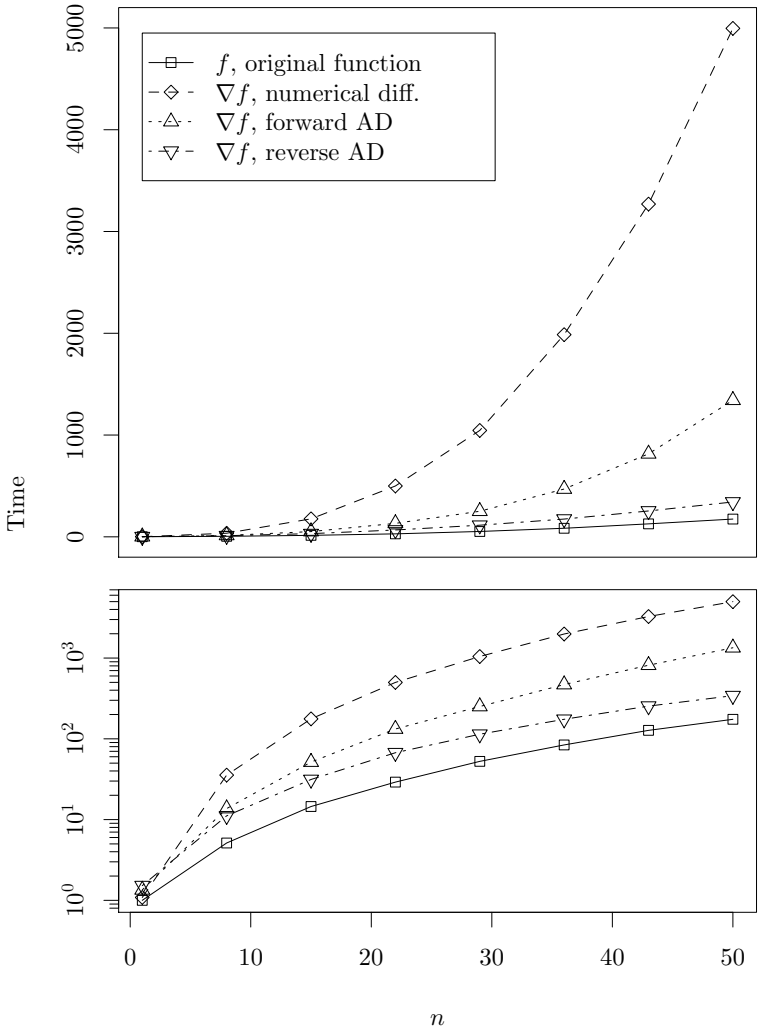


**Fig. 5** Evaluation time of the Helmholtz free energy function of a mixed fluid, based on the Peng-Robinson equation of state (Peng and Robinson, 1976), $f(\mathbf{x}) = RT\sum_{i=0}^{n} \log \frac{x_i}{1-\mathbf{b^T x}} - \frac{\mathbf{x^T A x}}{\sqrt{8}\mathbf{b^T x}} \log \frac{1+(1+\sqrt{2})\mathbf{b^T x}}{1+(1-\sqrt{2})\mathbf{b^T x}}$, where $R$ is the universal gas constant, $T$ is the absolute temperature, $\mathbf{b} \in \mathbb{R}^n$ is a vector of constants, $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a symmetric matrix of constants, and $\mathbf{x} \in \mathbb{R}^n$ is the vector of independent variables describing the system. The plots show the evaluation time of $f$ and the gradient $\nabla f$ with numerical differentiation (central difference), forward mode AD, and reverse mode AD, as a function of the number of variables $n$. Reported times are relative to the evaluation time of $f$ with $n = 1$. Lower figure shows the data with a logarithmic scale for illustrating the behavior when $n < 20$. Numerical results are given in Table 4. (Code available online: http://gbaydin.github.io/DiffSharp/examples-helmholtzenergyfunction.html)

ular such method is the BFGS[11] algorithm, together with its limited-memory variant L-BFGS (Dennis and Schnabel, 1996).

AD here provides a way of computing the exact Hessian in an efficient way[12]. However, in many cases, one does not need the full Hessian but only a Hessian-vector product $\mathbf{H}\,\mathbf{r}$, which can be computed very efficiently using a combination of the forward and reverse modes of AD[13]. This computes $\mathbf{H}\,\mathbf{r}$ with $O(n)$ complexity, even though the $\mathbf{H}$ is a $n \times n$ matrix. Moreover, Hessians arising in large-scale applications are typically sparse. This sparsity, along with symmetry, can be readily exploited by AD techniques such as computational graph elimination (Dixon, 1991), partial separability (Gay, 1996), and matrix coloring and compression (Gebremedhin et al, 2009).

Another approach for improving the rate of convergence of gradient methods is to use gain adaptation methods such as stochastic meta-descent (SMD) (Schraudolph, 1999), where stochastic sampling is introduced to avoid local minima. An example using SMD with AD Hessian-vector products is given by Vishwanathan et al (2006) on conditional random fields (CRF), a probabilistic method for labeling and segmenting data. Similarly, Schraudolph and Graepel (2003) use Hessian-vector products in their model combining conjugate gradient techniques with stochastic gradient descent.

### 4.2 Neural networks

Training of neural networks is an optimization problem with respect to a set of weights, which can in principle be addressed via any method including gradient descent, stochastic gradient descent (Zhenzhen and Elhanany, 2007), or BFGS (Apostolopoulou et al, 2009). As we have seen, the highly successful backpropagation algorithm is only a specialized version of reverse mode AD: by applying the reverse mode to any algorithm evaluating a network's error as a function of its weights, we can readily compute the partial derivatives needed for performing weight updates[14].

There are instances in neural network literature—albeit few—where explicit reference is made to AD for computing error gradients, such as Eriksson et al (1998) using AD for large-scale feed-forward networks, and the work by Yang et al (2008), where they use AD to train a neural network-based proportional-integral-derivative (PID) controller. Similarly, Rollins (2009) uses reverse mode AD in conjunction with neural networks for the problem of optimal feedback control.

[11] After Broyden–Fletcher–Goldfarb–Shanno, who independently discovered the method in the 1970s.

[12] See `http://gbaydin.github.io/DiffSharp/examples-newtonsmethod.html` for an implementation of Newton's method with the full Hessian.

[13] For example, by applying the reverse mode to gradient code already produced through the forward mode. That is, given the function $f : \mathbb{R}^n \to \mathbb{R}$, the evaluation point $\mathbf{x}$, and the vector $\mathbf{r}$, first computing the directional derivative $\nabla f \cdot \mathbf{r}$ through the forward mode via setting $\dot{\mathbf{x}} = \mathbf{r}$ and then applying the reverse mode on this result to get $\nabla^2 f \cdot \mathbf{r}$.

[14] See `http://gbaydin.github.io/DiffSharp/examples-neuralnetworks.html` for an implementation of backpropagation with reverse mode AD.

Beyond backpropagation, the generality of AD also opens up other possibilities. An example is given for continuous time recurrent neural networks (CTRNN) by Al Seyab and Cao (2008), where they apply AD for the training of CTRNNs predicting dynamic behavior of nonlinear processes in real time. The authors use AD for computing derivatives higher than second-order and report significantly reduced network training time compared with other methods.

### 4.3 Computer vision and image processing

In image processing, 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 image coordinates, approximating those derived on a hypothetical continuous image space. As a consequence, derivatives are approximated using numerical differences.

On the other hand, in computer vision, many problems are formulated as the minimization of an appropriate energy functional (Bertero et al, 1988; Chambolle, 2000). This minimization is usually accomplished via calculus of variations and the Euler-Lagrange equation. In this area, the first study introducing AD to computer vision is given by Pock et al (2007), where they address the problems of denoising, segmentation, and recovery of information from stereoscopic image pairs, and note the usefulness of AD in identifying sparsity patterns in large Jacobian and Hessian matrices.

In another study, Grabner et al (2008) use reverse mode AD for GPU-accelerated medical 2D/3D registration, a task involving the alignment of data from different sources such as X-ray images or computed tomography. The authors report a six-fold increase in speed compared with numerical differentiation using center difference (cf. our benchmark with the Helmholtz function, Fig. 5 and Table 4), demonstrating that the computer vision field is ripe for AD applications.

Barrett and Siskind (2013) present a use of AD for the task of video event detection. Compared with general computer vision tasks focused on recognizing objects and their properties (which can be thought of as *nouns* in a narrative), an important aspect of this work is that it aims to recognize and reason about events and actions (i.e. *verbs*). The method uses Hidden Markov Models (HMMs) and Dalal and Triggs (2005) object detectors, and performs training on a corpus of pre-tracked video by an adaptive step size naive gradient descent algorithm, where gradient computations are done with reverse mode AD. Initially implemented with the R6RS-AD package[15] providing forward and reverse mode AD in R6RS Scheme, the gradient code was later ported to C and highly optimized. Even if the final detection code does not di-

[15] https://github.com/qobi/R6RS-AD

rectly use AD, the authors report[16] that AD in this case served as a foundation and a correctness measure for validating subsequent work.

### 4.4 Natural language processing

Within the natural language processing (NLP) field, statistical models are commonly trained using general purpose or specialized gradient methods and mostly remain expensive to train. Improvements in training time can be realized by using online or distributed training algorithms (Gimpel et al, 2010). An example using stochastic gradient descent for NLP is given by Finkel et al (2008) optimizing conditional random field parsers through an objective function. Related with the work on video event detection in the previous section, Yu and Siskind (2013) report their work on sentence tracking, representing an instance of grounded language learning paired with computer vision, where the system learns word meanings from short video clips paired with descriptive sentences. The method works by using HMMs to represent changes in video frames and meanings of different parts of speech. This work is implemented in C and computes the required gradients using AD through the ADOL-C tool[17].

### 4.5 Probabilistic programming and Bayesian methods

Probabilistic programming has been experiencing a recent resurgence thanks to new algorithmic advances for probabilistic inference and new areas of application in machine learning (Goodman, 2013). A probabilistic programming language provides primitive language constructs for random choice and allows the automatic probabilistic inference of distributions specified by programs.

Inference techniques can be static, such as compiling programs to Bayesian networks and using algorithms such as belief propagation for inference; or they can be dynamic, executing programs several times and computing statistics on observed values to infer distributions. Markov chain Monte Carlo (MCMC) methods are typically used for dynamic inference, such as the Metropolis-Hastings algorithm based on random sampling. Meyer et al (2003) give an example of how AD can be used for speeding up Bayesian posterior inference in MCMC, with an application in stochastic volatility.

When model parameters are continuous, the Hamiltonian—or, hybrid—Monte Carlo (HMC) algorithm provides improved convergence characteristics avoiding the slow exploration of random sampling, by simulating Hamiltonian dynamics through auxiliary "momentum variables" (Neal, 1993).

The advantages of HMC come at the cost of requiring gradient evaluations of complex probability models. AD is highly suitable here for complementing probabilistic programming, because it relieves the user from the manual

[16] Through personal communication.

[17] An implementation of the sentence tracker applied to video search using sentence-based queries can be accessed online: http://upplysingaoflun.ecn.purdue.edu/~qobi/cccp/sentence-tracker-video-retrieval.html

computation of derivatives for each model. For instance, the probabilistic programming language Stan[18] implements automatic Bayesian inference based on HMC and the No-U-Turn sampler (NUTS) (Hoffman and Gelman, 2014) and uses reverse mode AD for the calculation of gradients for both HMC and NUTS. Similarly, Wingate et al (2011) demonstrate the use of AD as a non-standard interpretation of probabilistic programs enabling efficient inference algorithms.

AD is particularly promising in this domain because of the dynamic nature of probabilistic programs, that is, dynamically creating or deleting random variables and making it very difficult to formulate closed-form expressions for gradients.

## 5 Implementations

For picking the best tool for a particular application, it is useful to have an understanding of the different ways in which AD can be implemented. Here we cover major implementation strategies and provide a survey of existing tools.

A principal consideration in any AD implementation is the performance overhead introduced by the AD arithmetic and bookkeeping. In terms of computational complexity, AD ensures that the amount of arithmetic goes up by no more than a small constant. For instance, for the reverse mode, the extra arithmetic corresponds to at most four or five times the arithmetic needed to evaluate the original function[19]. But, managing this arithmetic can introduce a significant overhead if done carelessly. For instance, naively allocating data structures for holding dual numbers will involve memory access and allocation for every arithmetic operation, which are usually more expensive than arithmetic operations on modern computers. Likewise, using operator overloading may introduce method dispatches with attendant costs, which, compared to raw numerical computation of the original function, can easily amount to a slowdown of an order of magnitude.

Another major issue is the possibility of a class of bugs called "perturbation confusion" (Siskind and Pearlmutter, 2005). This essentially means that if two ongoing differentiations affect the same piece of code, the two formal epsilons they introduce (Sect. 3.1.1) need to be kept distinct. It is very easy to have bugs—in particularly performance-oriented AD implementations—that confuse these in various ways. Such situations can also arise when AD is nested, that is, derivatives are computed for functions that internally take derivatives.

One should be also cautious about approximated functions and AD. In this case, if you have a procedure *approximating* an ideal function, AD always gives the derivative of the procedure that was actually programmed, which may not be a good approximation of the derivative of the ideal function that

[18] http://mc-stan.org/

[19] See Griewank and Walther (2008) for a detailed analysis of forward and reverse mode complexity bounds.

the procedure was approximating[20]. Users of AD implementations must be therefore cautious to *approximate the derivative, not differentiate the approximation*. This would require explicitly approximating a known derivative, in cases where a mathematical function can only be computed approximately but has a well-defined mathematical derivative.

In conjunction with Table 5, we present a review of notable AD implementations[21]. A thorough taxonomy of implementation techniques was introduced by Juedes (1991), which was later revisited by Bischof et al (2008) and simplified into *elemental*, *operator overloading*, *compiler-based*, and *hybrid* methods. We adopt a similar classification for briefly presenting the currently popular tools.

### 5.1 Elemental libraries

These implementations form the most basic category and work by replacing mathematical operations with calls to an AD-enabled library. Methods exposed by the library are then used in function definitions, meaning that the decomposition of any function into elementary operations is done manually at the same time with writing the code.

The approach has been utilized since the early days of AD, prototypical examples being the WCOMP and UCOMP packages of Lawson (1971), the APL package of Neidinger (1989), and the work by Hinkins (1994). Likewise, Hill and Rich (1992) formulate their implementation of AD in MATLAB using elemental methods.

Elemental methods still constitute the simplest strategy to implement AD for languages without operator loading.

### 5.2 Compilers and source transformation

These implementations provide extensions to programming languages that automate the decomposition of equations into AD-enabled elementary operations. They are typically executed as preprocessors[22] to transform the input in the extended language into the original language.

Classical instances of source code transformation include the Fortran preprocessors GRESS (Horwedel et al, 1988) and PADRE2 (Kubo and Iri, 1990), which transform AD-enabled variants of Fortran into standard Fortran 77 before compiling. Similarly, the ADIFOR tool by Bischof et al (1996), given a

[20] As an example, consider $e^x$ computed by a piecewise-rational approximation routine. Using AD on this routine would produce an approximated derivative in which each piece of the piecewise formula will get differentiated. Even if this would remain an approximation of the derivative of $e^x$, we know that $\frac{de^x}{dx} = e^x$ and the original approximation itself was already a better approximation for the derivative of $e^x$. In modern computers this is not an issue, because $e^x$ is a primitive implemented in hardware.

[21] Also see the website `http://www.autodiff.org/` for a list of tools maintained by the AD community.

[22] Preprocessors transform program source code before it is given as an input to a compiler.

**Table 5** Survey of major AD implementations.

| Language | Tool | Type | Mode | Institution / Project | References | URL |
|---|---|---|---|---|---|---|
| **AMPL** | AMPL | INT | F, R | Bell Laboratories | Fourer et al (2002) | http://www.ampl.com/ |
| **C, C++** | ADIC | ST | F, R | Argonne National Laboratory | Bischof et al (1997) | http://www-new.mcs.anl.gov/adic/down-2.htm |
| | ADOL-C | OO | F, R | Computational Infrastructure for Operations Research | Walther and Griewank (2012) | http://www.coin-or.org/projects/ADOL-C.xml |
| **C++** | Ceres Solver | LIB | F | Google | | http://ceres-solver.org/ |
| | CppAD | OO | F, R | Computational Infrastructure for Operations Research | Bell and Burke (2008) | http://www.coin-or.org/CppAD/ |
| | FADBAD++ | OO | F, R | Technical University of Denmark | Bendtsen and Stauning (1996) | http://www.fadbad.com/fadbad.html |
| | Mxyzptlk | OO | F | Fermi National Accelerator Laboratory | Ostiguy and Michelotti (2007) | https://cdcvs.fnal.gov/redmine/projects/fermitools/wiki/MXYZPTLK |
| **C#** | AutoDiff | LIB | R | George Mason Univ., Department of Computer Science | Shtof et al (2013) | http://autodiff.codeplex.com/ |
| **F#** | DiffSharp | LIB | F, R | National University of Ireland Maynooth | | http://gbaydin.github.io/DiffSharp/ |
| **Fortran** | ADIFOR | ST | F, R | Argonne National Laboratory | Bischof et al (1996) | http://www.mcs.anl.gov/research/projects/adifor/ |
| | NAGWare | COM | F, R | Numerical Algorithms Group | Naumann and Riehme (2005) | http://www.nag.co.uk/nagware/Research/ad_overview.asp |
| | TAMC | ST | R | Max Planck Institute for Meteorology | Giering and Kaminski (1998) | http://autodiff.com/tamc/ |
| **Fortran, C/C++** | COSY | INT | F | Michigan State Univ., Biomedical and Physical Sciences | Berz et al (1996) | http://www.bt.pa.msu.edu/index_cosy.htm |
| | Tapenade | ST | F, R | INRIA Sophia-Antipolis | Hascoët and Pascual (2013) | http://www-sop.inria.fr/tropics/tapenade.html |
| **Haskell** | ad | OO | F, R | Haskell package | | http://hackage.haskell.org/package/ad |
| **Java** | Deriva | LIB | F | Java & Clojure library | | https://github.com/lambder/Deriva |
| **MATLAB** | ADiMat | ST, OO | F, R | Technical University of Darmstadt, Scientific Computing | Willkomm and Vehreschild (2013) | http://adimat.sc.informatik.tu-darmstadt.de/ |
| | INTLab | OO | F | Hamburg University of Technology, Institute for Reliable Computing | Rump (1999) | http://www.ti3.tu-harburg.de/rump/intlab/ |
| | TOMLAB /MAD | OO | F | Cranfield University & Tomlab Optimization Inc. | Forth (2006) | http://tomlab.biz/products/mad |
| **Python** | ad | OO | R | Python package | | https://pypi.python.org/pypi/ad |
| **Scheme** | Scmutils | OO | F | MIT Computer Science and Artificial Intelligence Lab. | Sussman and Wisdom (2001) | http://groups.csail.mit.edu/mac/users/gjs/6946/refman.txt |

F: Forward, R: Reverse; COM: Compiler, INT: Interpreter, LIB: Library, OO: Operator overloading, ST: Source transformation

Fortran source code, generates an augmented code in which all specified partial derivatives are computed in addition to the original result. For procedures coded in ANSI C, the ADIC tool (Bischof et al, 1997) implements AD as a source transformation after the specification of dependent and independent variables. A recent and popular tool also utilizing this approach is Tapenade (Pascual and Hascoët, 2008; Hascoët and Pascual, 2013), implementing forward and reverse mode AD for Fortran and C programs. Tapenade itself is implemented in Java and can be run locally or as an online service[23].

In addition to language extensions through source code transformation, there are implementations introducing new languages with tightly integrated AD capability through special-purpose compilers or interpreters. Some of the earliest AD tools such as SLANG (Adamson and Winant, 1969) and PROSE (Pfeiffer, 1987) belong to this category. The NAGWare Fortran 95 compiler (Naumann and Riehme, 2005) is a more recent example, where the use of AD-related extensions triggers automatic generation of derivative code at compile time.

As an example of interpreter-based implementation, the algebraic modeling language AMPL (Fourer et al, 2002) enables one to express objectives and constraints in mathematical notation, from which the system deduces active variables and arranges the necessary AD computations. Other examples in this category include the FM/FAD package (Mazourik, 1991), based on the Algol-like DIFALG language, and the object-oriented COSY language (Berz et al, 1996) similar to Pascal.

The Stalingrad compiler (Pearlmutter and Siskind, 2008), working on the Scheme-based AD-aware VLAD language, also falls under this category. The newer DVL compiler[24] is based on Stalingrad and uses a reimplementation of portions of the VLAD language.

### 5.3 Operator overloading

In modern programming languages with polymorphic features, operator overloading provides the most straightforward way of implementing AD, exploiting the capability of redefining elementary operation semantics.

A highly popular tool implemented with operator overloading in C++ is ADOL-C (Walther and Griewank, 2012). ADOL-C requires the use of AD-enabled types for variables and works via recording arithmetic operators on variables in data structures called "tapes", which can subsequently be "played back" during reverse mode AD computations. The Mxyzptlk package (Michelotti, 1990) is another example for C++ capable of computing arbitrary-order partial derivatives via forward propagation. The FADBAD++ library (Bendtsen and Stauning, 1996) implements AD for C++ using templates and oper-

[23] http://www-tapenade.inria.fr:8080/tapenade/index.jsp

[24] https://github.com/axch/dysvunctional-language

ator overloading. For Python, the *ad* package[25] uses operator overloading to compute first- and second-order derivatives.

For functional languages, examples include the AD routines within the Scmutils library[26] for Scheme, the *ad* library[27] for Haskell, and the recent DiffSharp library[28] for F#.

## 6 Conclusions

Given all its advantages, AD has remained remarkably underused by the machine learning community. We reason that this is mostly because it is poorly understood and frequently confused with the better known symbolic and numerical differentiation methods. In comparison, increasing awareness of AD in fields such as engineering design optimization (Hascoët et al, 2003), computational fluid dynamics (Müller and Cusdin, 2005), climatology (Charpentier and Ghemires, 2000), and computational finance (Bischof et al, 2002) provide evidence for its maturity and efficiency, with benchmarks (Giles and Glasserman, 2006; Sambridge et al, 2007; Capriotti, 2011) reporting performance increases of several orders of magnitude.

More often than not, machine learning articles tend to present the calculation of analytical derivatives for novel models as an important technical feat, potentially taking up as much space as the main contribution. Needless to say, there are occasions where we are interested in obtaining more than just the numerical value of derivatives. Derivative expressions can be useful for analysis and offer an 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.

The dependence on manual or symbolic differentiation impedes expressiveness of models by limiting the set of operations to those for which symbolic derivatives can be computed. Using AD, in contrast, enables us to build models using the full set of algorithmic machinery, knowing that we will be able to compute exact derivatives without handling parts of the model as closed-form expressions.

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

[25] http://pythonhosted.org/ad/

[26] http://groups.csail.mit.edu/mac/users/gjs/6946/refman.txt

[27] http://hackage.haskell.org/package/ad

[28] http://gbaydin.github.io/DiffSharp/

## References

Adamson DS, Winant CW (1969) A SLANG simulation of an initially strong shock wave downstream of an infinite area change. In: Proceedings of the Conference on Applications of Continuous-System Simulation Languages, pp 231–40

Al Seyab RK, Cao Y (2008) Nonlinear system identification for predictive control using continuous time recurrent neural networks and automatic differentiation. Journal of Process Control 18(6):568–581, DOI 10.1016/j.jprocont.2007.10.012

Apostolopoulou MS, Sotiropoulos DG, Livieris IE, Pintelas P (2009) A memoryless BFGS neural network training algorithm. In: 7th IEEE International Conference on Industrial Informatics, INDIN 2009, pp 216–221, DOI 10.1109/INDIN.2009.5195806

Barrett DP, Siskind JM (2013) Felzenszwalb-Baum-Welch: Event detection by changing appearance. arXiv preprint arXiv:13064746

Bauer FL (1974) Computational graphs and rounding error. SIAM Journal on Numerical Analysis 11(1):87–96

Beda LM, Korolev LN, Sukkikh NV, Frolova TS (1959) Programs for automatic differentiation for the machine BESM (in russian)

Bell BM, Burke JV (2008) Algorithmic differentiation of implicit functions and optimal values. In: Bischof CH, Bücker HM, Hovland P, Naumann U, Utke J (eds) Advances in Automatic Differentiation, Lecture Notes in Computational Science and Engineering, vol 64, Springer Berlin Heidelberg, pp 67–77, DOI 10.1007/978-3-540-68942-3_7

Bendtsen C, Stauning O (1996) FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM-REP-1996-17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark

Bert CW, Malik M (1996) Differential quadrature method in computational mechanics: A review. Applied Mechanics Reviews 49, DOI 10.1115/1.3101882

Bertero M, Poggio T, Torre V (1988) Ill-posed problems in early vision. Proceedings of the IEEE 76(8):869–89

Berz M, Makino K, Shamseddine K, Hoffstätter GH, Wan W (1996) COSY INFINITY and its applications in nonlinear dynamics. In: Berz M, Bischof C, Corliss G, Griewank A (eds) Computational Differentiation: Techniques, Applications, and Tools, Society for Industrial and Applied Mathematics, Philadelphia, PA, pp 363–5

Bischof C, Khademi P, Mauer A, Carle A (1996) ADIFOR 2.0: Automatic differentiation of fortran 77 programs. Computational Science Engineering, IEEE 3(3):18–32, DOI 10.1109/99.537089

Bischof C, Roh L, Mauer A (1997) ADIC: An extensible automatic differentiation tool for ANSI-C. Software Practice and Experience 27(12):1427–56

Bischof CH, Bücker HM, Lang B (2002) Automatic differentiation for computational finance. In: Kontoghiorghes EJ, Rustem B, Siokos S (eds) Computational Methods in Decision-Making, Economics and Finance, Applied Optimization, vol 74, Springer US, pp 297–310, DOI 10.1007/978-1-4757-3613-7_15

Bischof CH, Hovland PD, Norris B (2008) On the implementation of automatic differentiation tools. Higher-Order and Symbolic Computation 21(3):311–31, DOI 10.1007/s10990-008-9034-4

Bottou L (1998) Online learning and stochastic approximations. On-line learning in neural networks 17:9

Brezinski C, Zaglia MR (1991) Extrapolation Methods: Theory and Practice. North-Holland

Bryson AE Jr (1962) A steepest ascent method for solving optimum programming problems. Journal of Applied Mechanics 29(2):247

Bryson AE Jr, Ho YC (1969) Applied optimal control. Blaisdell, Waltham, MA

Burden RL, Faires JD (2001) Numerical Analysis. Brooks/Cole

Capriotti L (2011) Fast greeks by algorithmic differentiation. Journal of Computational Finance 14(3):3

Carmichael GR, Sandu A (1997) Sensitivity analysis for atmospheric chemistry models via automatic differentiation. Atmospheric Environment 31(3):475–89

Chambolle A (2000) Inverse problems in image processing and image segmentation: some mathematical and numerical aspects. Springer Lecture Notes

Charpentier I, Ghemires M (2000) Efficient adjoint derivatives: application to the meteorological model meso-nh. Optimization Methods and Software 13(1):35–63

Clifford WK (1873) Preliminary sketch of bi-quaternions. Proceedings of the London Mathematical Society 4:381–95

Corliss GC (1988) Application of differentiation arithmetic, Perspectives in Computing, vol 19, Academic Press, Boston, pp 127–48

Dalal N, Triggs B (2005) Histograms of oriented gradients for human detection. In: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), IEEE Computer Society, Washington, DC, USA, pp 886–93, DOI 10.1109/CVPR.2005.177

Dauvergne B, Hascoët L (2006) The data-flow equations of checkpointing in reverse automatic differentiation. In: Alexandrov VN, van Albada GD, Sloot PMA, Dongarra J (eds) Computational Science – ICCS 2006, Springer Berlin, Dauvergne, Lecture Notes in Computer Science, vol 3994, pp 566–73

Dennis JE, Schnabel RB (1996) Numerical Methods for Unconstrained Optimization and Nonlinear Equations. Classics in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia

Dixon LC (1991) Use of automatic differentiation for calculating hessians and newton steps. In: Griewank A, Corliss GF (eds) Automatic Differentiation of Algorithms: Theory, Implementation, and Application, SIAM, Philadelphia, PA, pp 114–125

Draper NR, Smith H (1998) Applied Regression Analysis. Wiley-Interscience

Ekström U, Visscher L, Bast R, Thorvaldsen AJ, Ruud K (2010) Arbitrary-order density functional response theory from automatic differentiation. Journal of Chemical Theory and Computation 6:1971–80, DOI 10.1021/ct100117s

Eriksson J, Gulliksson M, Lindström P, Wedin P (1998) Regularization tools for training large feed-forward neural networks using automatic differentiation. Optimization Methods and Software 10(1):49–69, DOI 10.1080/10556789808805701

Finkel JR, Kleeman A, Manning CD (2008) Efficient, feature-based, conditional random field parsing. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL 2008), pp 959–67

Fornberg B (1981) Numerical differentiation of analytic functions. ACM Transactions on Mathematical Software 7(4):512–26, DOI 10.1145/355972.355979

Forth SA (2006) An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. ACM Transactions on Mathematical Software 32(2):195–222

Fourer R, Gay DM, Kernighan BW (2002) AMPL: A Modeling Language for Mathematical Programming. Duxbury Press

Gay DM (1996) Automatically finding and exploiting partially separable structure in nonlinear programming problems

Gebremedhin A, Pothen A, Tarafdar A, Walther A (2009) Efficient computation of sparse hessians using coloring and automatic differentiation. INFORMS Journal on Computing 21(2):209–23, DOI 10.1287/ijoc.1080.0286

Giering R, Kaminski T (1998) Recipes for adjoint code construction. ACM Transactions on Mathematical Software 24:437–74, DOI 10.1145/293686.293695

Giles M, Glasserman P (2006) Smoking adjoints: fast monte carlo greeks. RISK 19(1):88–92

Gimpel K, Das D, Smith NA (2010) Distributed asynchronous online learning for natural language processing. In: Proceedings of the Fourteenth Conference on Computational Natural Language Learning, Association for Computational Linguistics, Stroudsburg, PA, USA, CoNLL '10, pp 213–222

Goltyanskii VG, Gamkrelidze RV, Pontryagin LS (1960) The theory of optimal processes I: The maximum principle. Invest Akad Nauk SSSR Ser Mat 24:3–42

Goodman ND (2013) The principles and practice of probabilistic programming. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, pp 399–402, DOI 10.1145/2429069.2429117

Gori M, Maggini M (1996) Optimal convergence of on-line backpropagation. IEEE Transactions on Neural Networks 7, DOI 10.1109/72.478415

Grabmeier J, Kaltofen E, Weispfenning VB (2003) Computer Algebra Handbook: Foundations, Applications, Systems. Springer

Grabner M, Pock T, Gross T, Kainz B (2008) Automatic differentiation for gpu-accelerated 2d/3d registration. In: Bischof CH, Bücker HM, Hovland P, Naumann U, Utke J (eds) Advances in Automatic Differentiation, Lecture Notes in Computational Science and Engineering, vol 64, Springer Berlin Heidelberg, pp 259–269, DOI 10.1007/978-3-540-68942-3_23

Griewank A (1989) On automatic differentiation. In: Iri M, Tanabe K (eds) Mathematical Programming: Recent Developments and Applications, Kluwer Academic Publishers, pp 83–108

Griewank A (2003) A mathematical view of automatic differentiation. Acta Numerica 12:321–98, DOI 10.1017/S0962492902000132

Griewank A (2012) Who invented the reverse mode of differentiation? Documenta Matematica, Extra Volume ISMP:389–400

Griewank A, Walther A (2008) Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Society for Industrial and Applied Mathematics, Philadelphia, DOI 10.1137/1.9780898717761

Hascoët L, Pascual V (2013) The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. ACM Transactions on Mathematical Software 39(3), DOI 10.1145/2450153.2450158

Hascoët L, Vázquez M, Dervieux A (2003) Automatic differentiation for optimum design, applied to sonic boom reduction. In: Kumar V, Gavrilova ML, Tan CJK, L'Ecuyer P (eds) Computational Science and Its Applications — ICCSA 2003, Lecture Notes in Computer Science, vol 2668, Springer Berlin Heidelberg, pp 85–94, DOI 10.1007/3-540-44843-8_10

Hecht-Nielsen R (1989) Theory of the backpropagation neural network. In: International Joint Conference on Neural Networks, IJCNN 1989, IEEE, pp 593–605

Hill DR, Rich LC (1992) Automatic differentiation in MATLAB. Applied Numerical Mathematics 9:33–43

Hinkins RL (1994) Parallel computation of automatic differentiation applied to magnetic field calculations. Tech. rep., Lawrence Berkeley Lab., CA

Hoffman MD, Gelman A (2014) The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. Journal of Machine Learning Research 15:1593–1623

Horwedel JE, Worley BA, Oblow EM, Pin FG (1988) GRESS version 1.0 user's manual. Technical Memorandum ORNL/TM 10835, Martin Marietta Energy Systems, Inc., Oak Ridge National Laboratory, Oak Ridge

Jerrell ME (1997) Automatic differentiation and interval arithmetic for estimation of disequilibrium models. Computational Economics 10(3):295–316

Juedes DW (1991) A taxonomy of automatic differentiation tools. In: Griewank A, Corliss GF (eds) Automatic Differentiation of Algorithms: Theory, Implementation, and Application, Society for Industrial and Applied Mathematics, Philadelphia, PA, pp 315–29

Kubo K, Iri M (1990) PADRE2, version 1—user's manual. Research Memorandum RMI 90-01, Department of Mathematical Engineering and Information Physics, University of Tokyo, Tokyo

Lawson CL (1971) Computing derivatives using W-arithmetic and U-arithmetic. Internal Computing Memorandum CM-286, Jet Propulsion Laboratory, Pasadena, CA

Leibniz GW (1685) Machina arithmetica in qua non additio tantum et subtractio sed et multiplicatio nullo, diviso vero paene nullo animi labore peragantur. Hannover

Mazourik V (1991) Integration of automatic differentiation into a numeircal library for PC's'. In: Griewank A, Corliss GF (eds) Automatic Differentiation of Algorithms: Theory, Implementation, and Application, Society for Industrial and Applied Mathematics, Philadelphia, PA, pp 315–29

Meyer R, Fournier DA, Berg A (2003) Stochastic volatility: Bayesian computation using automatic differentiation and the extended kalman filter. Econometrics Journal 6(2):408–420, DOI 10.1111/1368-423X.t01-1-00116

Michelotti L (1990) MXYZPTLK: A practical, user-friendly C++ implementation of differential algebra: User's guide. Technical Memorandum FN-535, Fermi National Accelerator Laboratory, Batavia, IL

Müller JD, Cusdin P (2005) On the performance of discrete adjoint cfd codes using automatic differentiation. International Journal for Numerical Methods in Fluids 47(8-9):939–945, DOI 10.1002/fld.885

Naumann U, Riehme J (2005) Computing adjoints with the NAGWare Fortran 95 compiler. In: Bücker HM, Corliss G, Hovland P, Naumann U, Norris B (eds) Automatic Differentiation: Applications, Theory, and Implementations, Lecture Notes in Computational Science and Engineering, Springer, pp 159–69

Neal R (1993) Probabilistic inference using markov chain monte carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto

Neal R (2011) Mcmc for using hamiltonian dynamics. Handbook of Markov Chain Monte Carlo pp 113–62, DOI 10.1201/b10905-6

Neidinger RD (1989) Automatic differentiation and APL. College Mathematics Journal 20(3):238–51, DOI 10.2307/2686776

Nolan JF (1953) Analytical differentiation on a digital computer. Master's thesis

Ostiguy JF, Michelotti L (2007) Mxyzptlk: An efficient, native C++ differentiation engine. In: Particle Accelerator Conference (PAC 2007)), IEEE, pp 3489–91, DOI 10.1109/PAC.2007.4440468

Parker DB (1985) Learning-logic. Tech. Rep. TR-47, Center for Computational Research in Economics and Management Science, MIT

Parker JR (2010) Algorithms for image processing and computer vision. Wiley

Pascual V, Hascoët L (2008) TAPENADE for C. In: Advances in Automatic Differentiation, Springer, Lecture Notes in Computational Science and Engineering, pp 199–210, DOI 10.1007/978-3-540-68942-3_18

Pearlmutter BA, Siskind JM (2008) Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM Transactions on Programming Languages and Systems 30(2):7:1–7:36, DOI 10.1145/1330017.1330018

Peng DY, Robinson DB (1976) A new two-constant equation of state. Industrial and Engineering Chemistry Fundamentals 15(1):59–64, DOI 10.1021/i160057a011

Pfeiffer FW (1987) Automatic differentiation in PROSE. SIGNUM Newsletter 22(1):2–8, DOI 10.1145/24680.24681

Pock T, Pock M, Bischof H (2007) Algorithmic differentiation: Application to variational problems in computer vision. IEEE Transactions on Pattern Analysis and Machine Intelligence 29(7):1180–1193, DOI 10.1109/TPAMI.2007.1044

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2007) Numerical Recipes: The Art of Scientific Computing. Cambridge University Press

Rall LB (2006) Perspectives on automatic differentiation: Past, present, and future? In: Bücker M, Corliss G, Naumann U, Hovland P, Norris B (eds) Automatic Differentiation: Applications, Theory, and Implementations, Lecture Notes in Computational Science and Engineering, vol 50, Springer Berlin Heidelberg, pp 1–14

Rollins E (2009) Optimization of neural network feedback control systems using automatic differentiation. Master's thesis, DOI 1721.1/59691

Rozonoer LI, Pontryagin LS (1959) Maximum principle in the theory of optimal systems I. Automation Remote Control 20:1288–302

Rumelhart DE, , Hinton GE, McClelland JL (1986) A general framework for parallel distributed processing, MIT Press, Cambridge, MA

Rump SM (1999) INTLAB—INTerval LABoratory. In: Developments in Reliable Computing, Kluwer Academic Publishers, Dordrecht, pp 77–104, DOI 10.1007/978-94-017-1247-7_7

Russ JC (2010) The Image Processing Handbook. CRC press

Sambridge M, Rickwood P, Rawlinson N, Sommacal S (2007) Automatic differentiation in geophysical inverse problems. Geophysical Journal International 170(1):1–8, DOI 10.1111/j.1365-246X.2007.03400.x

Schraudolph NN (1999) Local gain adaptation in stochastic gradient descent. In: Proceedings of the International Conference on Artificial Neural Networks, IEE London, Edinburgh, Scotland, pp 569–74, DOI 10.1049/cp:19991170

Schraudolph NN, Graepel T (2003) Combining conjugate direction methods with stochastic approximation of gradients. In: Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics

Shtof A, Agathos A, Gingold Y, Shamir A, Cohen-Or D (2013) Geosemantic snapping for sketch-based modeling. Computer Graphics Forum 32(2):245–53, DOI 10.1111/cgf.12044

Siskind JM, Pearlmutter BA (2005) Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In: Butterfield A (ed) Implementation and Application of Functional Languages—17th International Workshop, IFL'05, Dublin, Ireland, pp 1–9, trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60

Snyman JA (2005) Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms. Springer

Speelpenning B (1980) Compiling fast partial derivatives of functions given by algorithms. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign

Sra S, Nowozin S, Wright SJ (2011) Optimization for Machine Learning. MIT Press

Sussman GJ, Wisdom J (2001) Structure and Interpretation of Classical Mechanics. MIT Press, DOI 10.1063/1.1457268

Verma A (2000) An introduction to automatic differentiation. Current Science 78(7):804–7

Vishwanathan SVN, Schraudolph NN, Schmidt MW, Murphy KP (2006) Accelerated training of conditional random fields with stochastic gradient methods. In: Proceedings of the 23rd international conference on Machine learning (ICML '06), pp 969–76, DOI 10.1145/1143844.1143966

Walther A (2007) Automatic differentiation of explicit Runge-Kutta methods for optimal control. Computational Optimization and Applications 36(1):83–108, DOI 10.1007/s10589-006-0397-3

Walther A, Griewank A (2012) Getting started with ADOL-C. In: Naumann U, Schenk O (eds) Combinatorial Scientific Computing, Chapman-Hall CRC Computational Science, chap 7, pp 181–202, DOI 10.1201/b11644-8

Wengert R (1964) A simple automatic derivative evaluation program. Communications of the ACM 7:463–4

Werbos PJ (1974) Beyond regression: New tools for prediction and analysis in the behavioral sciences. PhD thesis, Harvard University

Widrow B, Lehr MA (1990) 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. Proceedings of the IEEE 78(9):1415–42, DOI 10.1109/5.58323

Willkomm J, Vehreschild A (2013) The ADiMat handbook. URL http://adimat.sc.informatik.tu-darmstadt.de/doc/

Wingate D, Goodman ND, Stuhlmüller A, Siskind JM (2011) Nonstandard interpretations of probabilistic programs for efficient inference. Advances in Neural Information Processing Systems 23

Yang W, Zhao Y, Yan L, Chen X (2008) Application of PID controller based on BP neural network using automatic differentiation method. In: Sun F, Zhang J, Tan Y, Cao J, Yu W (eds) Advances in Neural Networks - ISNN 2008, Lecture Notes in Computer Science, vol 5264, Springer Berlin Heidelberg, pp 702–711, DOI 10.1007/978-3-540-87734-9_80

Yu H, Siskind JM (2013) Grounded language learning from video described with sentences. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Sofia, Bulgaria, pp 53–63

Zhenzhen L, Elhanany I (2007) Fast and scalable recurrent neural network learning based on stochastic meta-descent. In: American Control Conference, ACC 2007, pp 5694–5699, DOI 10.1109/ACC.2007.4282777

Zhu C, Byrd RH, Lu P, Nocedal J (1997) Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. ACM Transactions on Mathematical Software (TOMS) 23(4):550–60, DOI 10.1145/279232.279236