Journal of Machine Learning Research 18 (2018) 1-43 Submitted 8/17; Published 4/18

Automatic Differentiation in Machine Learning: a Survey

Atılım Güneş Baydin GUNES@ROBOTS.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford OX1 3PJ, United Kingdom

Barak A. Pearlmutter BARAK@PEARLMUTTER.NET
Department of Computer Science
National University of Ireland Maynooth
Maynooth, Co. Kildare, Ireland

Alexey Andreyevich Radul AXCH@MIT.EDU
Department of Brain and Cognitive Sciences
Massachusetts Institute of Technology
Cambridge, MA 02139, United States

Jeffrey Mark Siskind QOBI@PURDUE.EDU
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, United States

Editor: Léon Bottou

Abstract

Derivatives, mostly in the form of gradients and Hessians, are ubiquitous in machine learning. Automatic differentiation (AD), also called algorithmic differentiation or simply "autodiff", is a family of techniques similar to but more general than backpropagation for efficiently and accurately evaluating derivatives of numeric functions expressed as computer programs. AD is a small but established field with applications in areas including computational fluid dynamics, atmospheric sciences, and engineering design optimization. Until very recently, the fields of machine learning and AD have largely been unaware of each other and, in some cases, have independently discovered each other's results. Despite its relevance, general-purpose AD has been missing from the machine learning toolbox, a situation slowly changing with its ongoing adoption under the names "dynamic computational graphs" and "differentiable programming". We survey the intersection of AD and machine learning, cover applications where AD has direct relevance, and address the main implementation techniques. By precisely defining the main differentiation techniques and their interrelationships, we aim to bring clarity to the usage of the terms "autodiff", "automatic differentiation", and "symbolic differentiation" as these are encountered more and more in machine learning settings.

Keywords: Backpropagation, Differentiable Programming

1. Introduction

Methods for the computation of derivatives in computer programs can be classified into four categories: (1) manually working out derivatives and coding them; (2) *numerical differentiation* using finite difference approximations; (3) *symbolic differentiation* using expression manipulation in computer algebra systems such as Mathematica, Maxima, and Maple; and (4) *automatic differentiation*, also called *algorithmic differentiation*, which is the subject matter of this paper.

Conventionally, many methods in machine learning have required the evaluation of derivatives and most of the traditional learning algorithms have relied on the computation of gradients and Hessians of an objective function (Sra et al., 2011). When introducing new models, machine learning researchers have spent considerable effort on the manual derivation of analytical derivatives to subsequently plug these into standard optimization procedures such as L-BFGS (Zhu et al., 1997) or stochastic gradient descent (Bottou, 1998). Manual differentiation is time consuming and prone to error. Of the other alternatives, numerical differentiation is simple to implement but can be highly inaccurate due to round-off and truncation errors (Jerrell, 1997); more importantly, it scales poorly for gradients, rendering it inappropriate for machine learning where gradients with respect to millions of parameters are commonly needed. 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 models to be defined as closed-form expressions, ruling out or severely limiting algorithmic control flow and expressivity.

We are concerned with the powerful fourth technique, automatic differentiation (AD). AD performs a non-standard interpretation of a given computer program by replacing the domain of the variables to incorporate derivative values and redefining the semantics of the operators to propagate derivatives per the chain rule of differential calculus. Despite its widespread use in other fields, general-purpose AD has been underused by the machine learning community until very recently.[1] Following the emergence of deep learning (LeCun et al., 2015; Goodfellow et al., 2016) as the state-of-the-art in many machine learning tasks and the modern workflow based on rapid prototyping and code reuse in frameworks such as Theano (Bastien et al., 2012), Torch (Collobert et al., 2011), and TensorFlow (Abadi et al., 2016), the situation is slowly changing where projects such as autograd[2] (Maclaurin, 2016), Chainer[3] (Tokui et al., 2015), and PyTorch[4] (Paszke et al., 2017) are leading the way in bringing general-purpose AD to the mainstream.

The term "automatic" in AD can be a source of confusion, causing machine learning practitioners to put the label "automatic differentiation", or just "autodiff", on any method or tool that does not involve manual differentiation, without giving due attention to the underlying mechanism. We would like to stress that AD as a technical term refers to a specific family of techniques that compute derivatives through accumulation of values during code execution to generate numerical derivative evaluations rather than derivative

1. See, e.g., `https://justindomke.wordpress.com/2009/02/17/automatic-differentiation-the-most-criminally-underused-tool-in-the-potential-machine-learning-toolbox/`
2. `https://github.com/HIPS/autograd`
3. `https://chainer.org/`
4. `http://pytorch.org/`

expressions. This allows accurate evaluation of derivatives at machine precision with only a small constant factor of overhead and ideal asymptotic efficiency. In contrast with the effort involved in arranging code as closed-form expressions under the syntactic and semantic constraints of symbolic differentiation, AD can be applied to regular code with minimal change, allowing branching, loops, and recursion. Because of this generality, AD has been applied to computer simulations in industry and academia and found applications in fields including engineering design optimization (Forth and Evans, 2002; Casanova et al., 2002), computational fluid dynamics (Müller and Cusdin, 2005; Thomas et al., 2006; Bischof et al., 2006), physical modeling (Ekström et al., 2010), optimal control (Walther, 2007), structural mechanics (Haase et al., 2002), atmospheric sciences (Carmichael and Sandu, 1997; Charpentier and Ghemires, 2000), and computational finance (Bischof et al., 2002; Capriotti, 2011).

In machine learning, a specialized counterpart of AD known as the backpropagation algorithm has been the mainstay for training neural networks, with a colorful history of having been reinvented at various times by independent researchers (Griewank, 2012; Schmidhuber, 2015). 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 minima of an objective function. The required gradient is obtained by the backward propagation of the sensitivity of the objective value at the output (Figure 1), utilizing the chain rule to compute partial derivatives of the objective with respect to each weight. The resulting algorithm is essentially equivalent to transforming the network evaluation function composed with the objective function under reverse mode AD, which, as we shall see, actually generalizes the backpropagation idea. Thus, a modest understanding of the mathematics underlying backpropagation provides one with sufficient background for grasping AD techniques.

In this paper we review AD from a machine learning perspective, covering its origins, applications in machine learning, and methods of implementation. Along the way, we also aim to dispel some misconceptions that we believe have impeded wider recognition of AD by the machine learning community. In Section 2 we start by explicating how AD differs from numerical and symbolic differentiation. Section 3 gives an introduction to the AD technique and its forward and reverse accumulation modes. Section 4 discusses the role of derivatives in machine learning and examines cases where AD has relevance. Section 5 covers various implementation approaches and general-purpose AD tools, followed by Section 6 where we discuss future directions.

2. What AD Is Not

Without proper introduction, one might assume that AD is either a type of numerical or symbolic differentiation. Confusion can arise because AD does in fact provide numerical values of derivatives (as opposed to derivative expressions) and it does so by using symbolic rules of differentiation (but keeping track of derivative values as opposed to the resulting expressions), giving it a two-sided nature that is partly symbolic and partly numerical (Griewank, 2003). We start by emphasizing how AD is different from, and in several aspects superior to, these two commonly encountered techniques of computing derivatives.

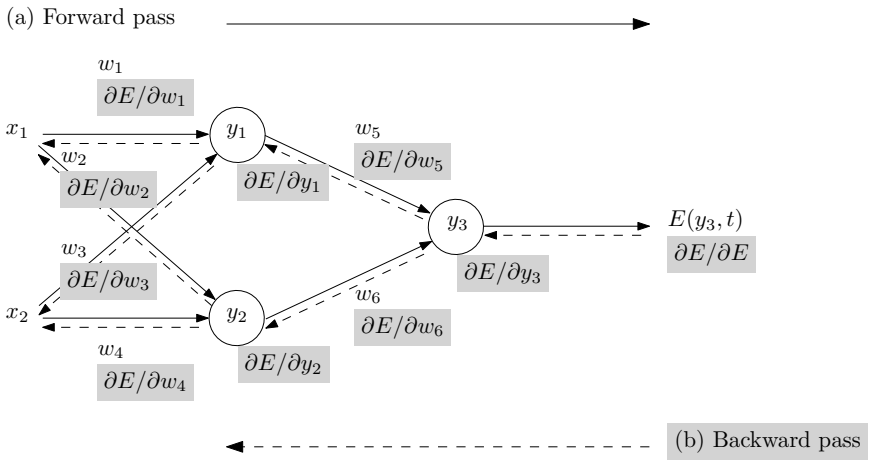

Figure 1: Overview of backpropagation. (a) Training inputs x_i are fed forward, generating corresponding activations y_i. An error E between the actual output y_3 and the target output t is computed. (b) The error adjoint is propagated backward, giving the gradient with respect to the weights $\nabla_{w_i} E = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_6}\right)$, which is subsequently used in a gradient-descent procedure. The gradient with respect to inputs $\nabla_{x_i} E$ can be also computed in the same backward pass.

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) (Figure 2, lower right). In its simplest form, it is based on the limit definition of a derivative. For example, for a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, one 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 $h > 0$ is a small step size. This has the advantage of being uncomplicated to implement, but the disadvantages of performing $O(n)$ evaluations of f for a gradient in n dimensions and requiring careful consideration in selecting the step size h.

Numerical approximations of derivatives are inherently ill-conditioned and unstable,[5] 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[6] and

5. 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"*.
6. 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.

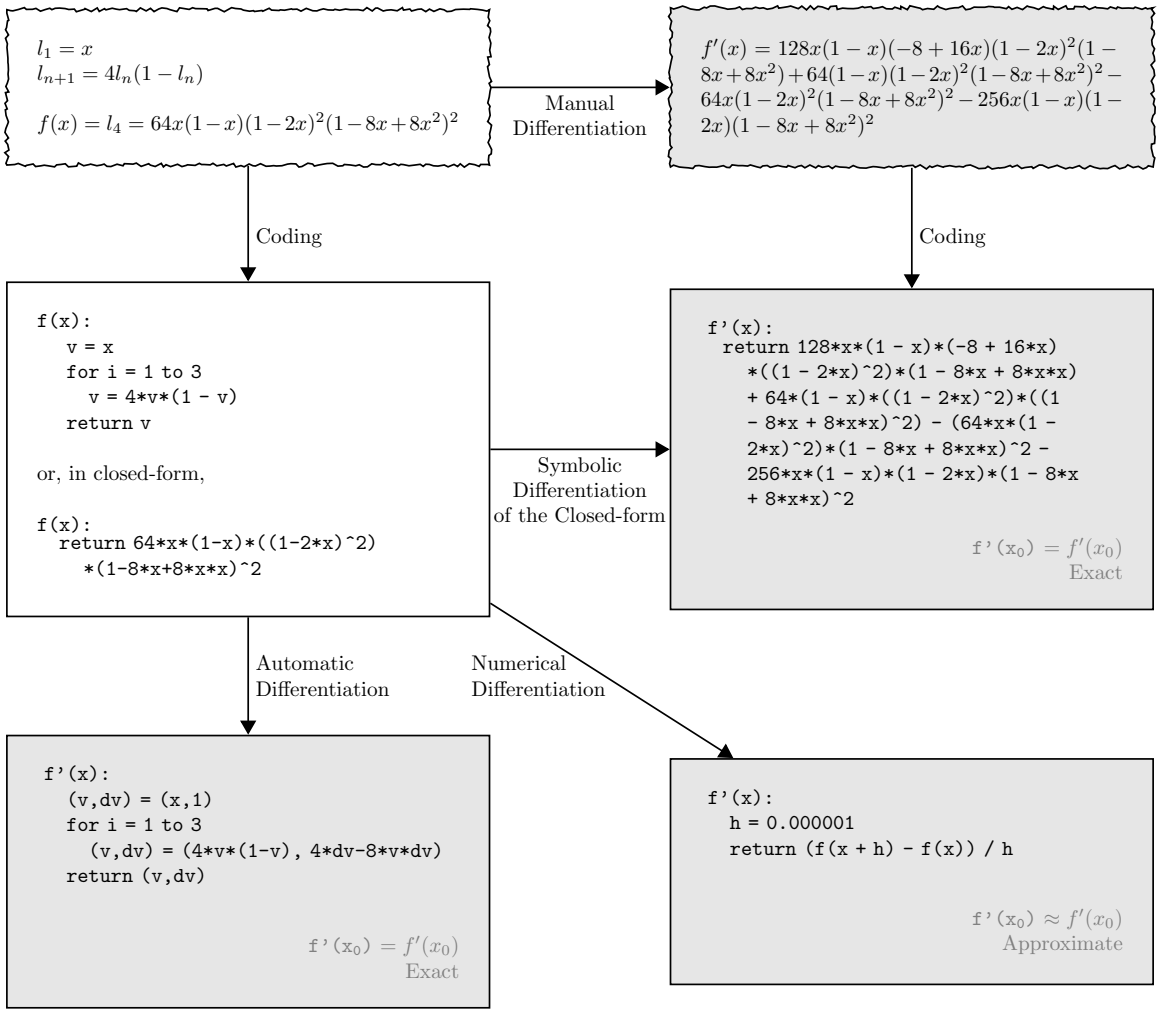

Figure 2: The range of approaches for differentiating mathematical expressions and computer code, looking at the example of a truncated logistic map (upper left). Symbolic differentiation (center right) gives exact results but requires closed-form input and suffers from expression swell; numerical differentiation (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 and support for control flow.

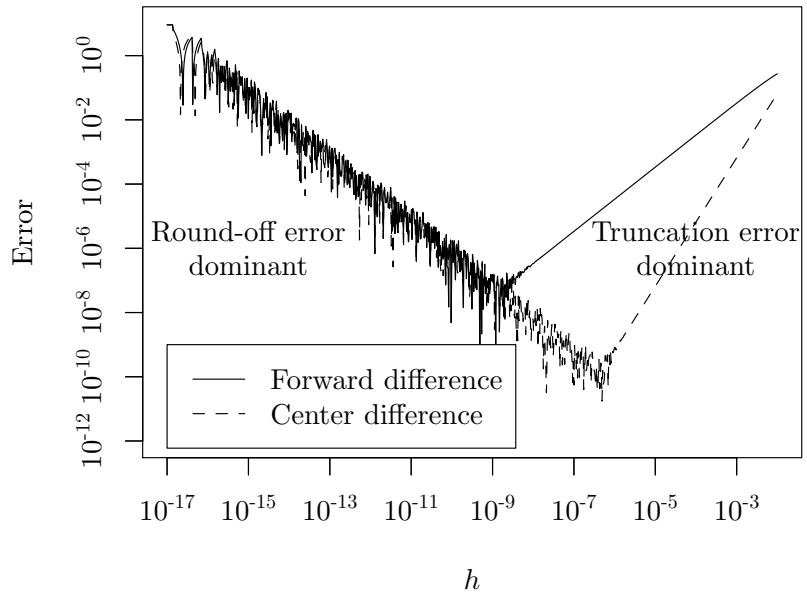

Figure 3: Error in the forward (Eq. 1) and center difference (Eq. 2) approximations as a function of step size h, for the derivative of the truncated logistic map $f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$. Plotted errors are computed using $E_{\text{forward}}(h, x_0) = \left| \frac{f(x_0+h)-f(x_0)}{h} - \frac{d}{dx} f(x)\big|_{x_0} \right|$ and $E_{\text{center}}(h, x_0) = \left| \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{d}{dx} f(x)\big|_{x_0} \right|$ at $x_0 = 0.2$.

round-off[7] 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 (Figure 3).

Various techniques have been developed to mitigate approximation errors in 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 one effectively moves the truncation error from first-order to second-order in h.[8] 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 dimensionality, a trade-off between accuracy and performance is faced, where computing a Jacobian matrix of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ requires $2mn$ evaluations.

7. 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.
8. This does not avoid either of the cardinal sins, and is still highly inaccurate due to truncation.

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), have increased computational complexity, do not completely eliminate approximation errors, and remain highly susceptible to floating point truncation.

The $O(n)$ complexity of numerical differentiation for a gradient in n dimensions is the main obstacle to its usefulness in machine learning, where n can be as large as millions or billions in state-of-the-art deep learning models (Shazeer et al., 2017). In contrast, approximation errors would be tolerated in a deep learning setting thanks to the well-documented error resiliency of neural network architectures (Gupta et al., 2015).

2.2 AD Is Not Symbolic Differentiation

Symbolic differentiation is the automatic manipulation of expressions for obtaining derivative expressions (Grabmeier and Kaltofen, 2003) (Figure 2, center right), carried out by applying transformations representing rules of differentiation such as

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

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

In optimization, symbolic derivatives can give valuable insight into the structure of the problem domain and, in some cases, produce analytical solutions of extrema (e.g., solving for $\frac{d}{dx}f(x)=0$) that can eliminate the need for derivative calculation altogether. On the other hand, symbolic derivatives do not lend themselves to efficient runtime calculation of derivative values, as they can get 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 $g(x)$. Note also that on the right hand side, $f(x)$ and $\frac{d}{dx}f(x)$ appear separately. If we just proceeded to symbolically differentiate $f(x)$ and plugged its derivative into the appropriate place, we would 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 numerical evaluation of derivatives and not so much with their actual symbolic form, it is in principle possible to significantly simplify computations by storing only the values of intermediate sub-expressions in memory. Moreover, for further efficiency, we can interleave as much as possible the differentiation and simplification steps. This interleaving idea forms the basis of AD and provides an account

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

of its simplest form: *apply symbolic differentiation at the elementary operation level and keep intermediate numerical results, in lockstep with the evaluation of the main function.* This is AD in the forward accumulation mode, which we shall introduce in the following section.

3. AD and Its Main Modes

AD can be thought of as performing a non-standard interpretation of a computer program where this interpretation involves augmenting the standard computation with the calculation of various derivatives. All numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known (Verma, 2000; Griewank and Walther, 2008), and combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition. Usually these elementary operations include the binary arithmetic operations, the unary sign switch, and transcendental functions such as the exponential, the logarithm, and the trigonometric functions.

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 (intermediate) variables, and
- variables $y_{m-i} = v_{l-i},\ i = m-1, \ldots, 0$ are the output variables.

Figure 4 shows the given trace of elementary operations represented as a computational graph (Bauer, 1974), useful in visualizing dependency relations between intermediate variables.

Evaluation traces form the basis of the AD techniques. An important point to note here is that AD can differentiate not only closed-form expressions in the classical sense, but also algorithms making use of control flow such as branching, loops, recursion, and procedure

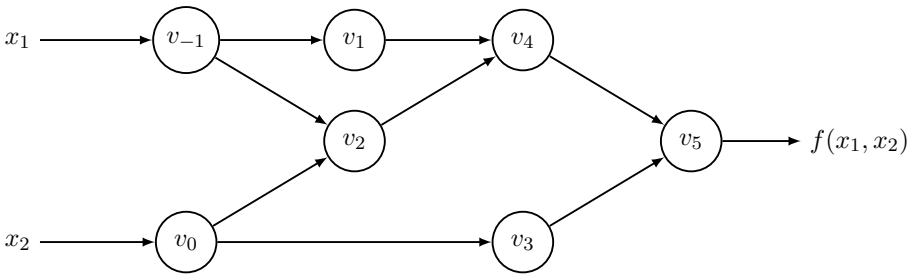

Figure 4: Computational graph of the example $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. See the primal trace in Tables 2 or 3 for the definitions of the intermediate variables $v_{-1} \dots v_5$.

calls, giving it an important advantage over symbolic differentiation which severely limits such expressivity. This is thanks to the fact that any numeric code will eventually result in a numeric evaluation trace with particular values of the input, intermediate, and output variables, which are the only things one needs to know for computing derivatives using chain rule composition, regardless of the specific control flow path that was taken during execution. Another way of expressing this is that AD is blind with respect to any operation, including control flow statements, which do not directly alter numeric values.

3.1 Forward Mode

AD in forward accumulation mode[9] is the conceptually most simple type. 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 in Table 2 and in graph form in Figure 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 primal trace, we generate the corresponding tangent (derivative) trace, given on the right-hand side in Table 2. Evaluating the primals v_i in lockstep with their corresponding tangents $\dot{v}_i$ 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 (input) variables x_i and m dependent (output) variables y_j. In this case, each forward pass of AD is initialized by setting only one of the variables $\dot{x}_i = 1$ and setting the rest to zero (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 $\mathbf{x} = \mathbf{a}$ then computes

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

9. Also called *tangent linear* mode.

Table 2: Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$ and setting $\dot{x}_1 = 1$ to compute $\frac{\partial y}{\partial x_1}$. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.

Forward Primal Trace			Forward Tangent (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. As a special case, when $f : \mathbb{R}^n \to \mathbb{R}$, we can obtain the directional derivative along a given vector $\mathbf{r}$ as a linear combination of the partial derivatives

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

by 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 $\frac{dy_i}{dx}$ can be computed with just one forward pass. Conversely, in the other extreme of $f : \mathbb{R}^n \to \mathbb{R}$, forward mode AD requires n evaluations to compute the gradient

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

which also corresponds to a $1 \times n$ Jacobian matrix that is built one column at a time with the forward mode in n evaluations.

In general, for cases $f : \mathbb{R}^n \to \mathbb{R}^m$ where $n \gg m$, a different technique is often preferred. We will describe AD in *reverse accumulation mode* in Section 3.2.

3.1.1 Dual Numbers

Mathematically, forward mode AD (represented by the left- and right-hand sides in Table 2) can be viewed as evaluating a function using dual numbers,[10] which can be defined as truncated Taylor series of the form

$$v + \dot{v}\epsilon \ ,$$

where $v, \dot{v} \in \mathbb{R}$ and ϵ is a nilpotent number such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. Observe, for example, that

$$\begin{aligned}(v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) &= (v + u) + (\dot{v} + \dot{u})\epsilon \\ (v + \dot{v}\epsilon)(u + \dot{u}\epsilon) &= (vu) + (v\dot{u} + \dot{v}u)\epsilon \ ,\end{aligned}$$

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

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

and using dual numbers as data structures for carrying the tangent value together with the primal.[11] The chain rule works as expected on this representation: two applications of Eq. 5 give

$$\begin{aligned}f(g(v + \dot{v}\epsilon)) &= f(g(v) + g'(v)\dot{v}\epsilon) \\ &= f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon \ .\end{aligned}$$

The coefficient of ϵ 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 by interpreting any non-dual number v as $v + 0\epsilon$ and evaluating the function in this non-standard way on an initial input with a coefficient 1 for ϵ:

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

This also extends to arbitrary program constructs, since dual numbers, as data types, can be contained in any data structure. As long as a dual number remains in a data structure with no arithmetic operations being performed on it, 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.

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

10. First introduced by Clifford (1873), with important uses in linear algebra and physics.
11. Just as the complex number written $x + yi$ is represented in the computer as a pair in memory (x, y) whose two slots are reals, the dual number written $x + \dot{x}\epsilon$ is represented as the pair $(x, \dot{x})$. Such pairs are sometimes called Argand pairs (Hamilton, 1837, p107 Eqs. (157) and (158)).

3.2 Reverse Mode

AD in the reverse accumulation mode[12] corresponds to a generalized backpropagation algorithm, in that it propagates derivatives backward from a given output. This is done by complementing 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. In the case of backpropagation, y would be a scalar corresponding to the error E (Figure 1).

In reverse mode AD, derivatives are computed in the second phase of a two-phase process. In the first phase, the original function code is run *forward*, populating intermediate variables v_i and recording the dependencies in the computational graph through a bookkeeping procedure. In the second phase, 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-hand side. 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 to the change in the output y. Taking the variable v_0 as an example, we see in Figure 4 that the only way it can affect y is through affecting v_2 and v_3, so its contribution to the change in y is given by

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

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

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

lined up with the lines in the forward trace from which these expressions originate.

After the forward pass on the left-hand side, we run the reverse pass 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 pass.

Compared with the straightforwardness of forward accumulation 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 mechanistic procedure propagating derivatives forward.

An important advantage of the reverse mode is that it is significantly less costly to evaluate (in terms of operation count) than the forward mode for functions with a large number of inputs. 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 passes of the forward mode needed for populating the same. Because machine learning practice principally involves the gradient of a scalar-valued objective with respect to a large number of parameters, this establishes the reverse mode, as opposed to the forward mode, as the mainstay technique in the form of the backpropagation algorithm.

12. Also called *adjoint* or *cotangent linear* mode.

Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

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

In general, for a function $f : \mathbb{R}^n \to \mathbb{R}^m$, if we denote the operation count to evaluate the original function by $\mathrm{ops}(f)$, the time it takes to calculate the $m \times n$ Jacobian by the forward mode is $n\, c\, \mathrm{ops}(f)$, whereas the same computation can be done via reverse mode in $m\, c\, \mathrm{ops}(f)$, where c is a constant guaranteed to be $c < 6$ and typically $c \sim [2, 3]$ (Griewank and Walther, 2008). That is to say, reverse mode AD performs better when $m \ll n$.

Similar to the matrix-free computation of Jacobian–vector products with forward mode (Eq. 4), 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 phase with $\bar{\mathbf{y}} = \mathbf{r}$.

The advantages of reverse mode AD, however, come with the cost of increased storage requirements growing (in the worst case) in proportion to the number of operations in the evaluated function. It is an active area of research to improve storage requirements in implementations by using advanced methods such as checkpointing strategies and data-flow analysis (Dauvergne and Hascoët, 2006; Siskind and Pearlmutter, 2017).

3.3 Origins of AD and Backpropagation

Ideas underlying AD 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 in 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.

Reverse mode AD and backpropagation have an intertwined history. The essence of the reverse mode, cast in a continuous-time formalism, is the Pontryagin maximum principle (Rozonoer, 1959; Boltyanskii et al., 1960). This method was understood in the control theory community (Bryson and Denham, 1962; Bryson and Ho, 1969) and cast in more formal terms with discrete-time variables topologically sorted in terms of dependency by Werbos (1974). Prior to Werbos, the work by Linnainmaa (1970, 1976) is often cited as the first published description of the reverse mode. Speelpenning (1980) subsequently introduced reverse mode AD as we know it, in the sense that he gave the first implementation that was actually automatic, 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) and the Parallel Distributed Processing (PDP) group. The PDP group became aware of Parker's work only after their own discovery; similarly, Werbos' work was not appreciated until it was found by Parker (Hecht-Nielsen, 1989). This tells us an interesting story of two highly interconnected research communities that have somehow also managed to stay detached during this foundational period.

For a thorough review of the development of AD, we advise readers to refer to Rall (2006). Interested readers are highly recommended to read Griewank (2012) for an investigation of the origins of the reverse mode and Schmidhuber (2015) for the same for backpropagation.

4. AD and Machine Learning

In the following, we examine the main uses of derivatives in machine learning and report on a selection of works where general-purpose AD, as opposed to just backpropagation, has been successfully applied in a machine learning context. Areas where AD has seen use include optimization, neural networks, computer vision, natural language processing, and probabilistic inference.

4.1 Gradient-Based Optimization

Gradient-based optimization is one of the pillars of machine learning (Bottou et al., 2016). 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 of the form $\Delta\mathbf{w} = -\eta\nabla f$, where $\eta > 0$ is a step size. Gradient-based methods make use of the fact that f decreases steepest if one goes in the direction of the negative gradient. The convergence rate of gradient-based methods is usually improved by adaptive step-size techniques that adjust the step size η on every iteration (Duchi et al., 2011; Schaul et al., 2013; Kingma and Ba, 2015).

Table 4: Evaluation times of the Helmholtz free energy function and its gradient (Figure 5). Times are given relative to that of the original function with both (1) $n = 1$ and (2) 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 machine with Intel Core i7-4785T 2.20 GHz CPU and 16 GB RAM, using DiffSharp 0.5.7. The evaluation time for the original function with $n = 1$ is 0.0023 ms.

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

As we have seen, for large n, reverse mode AD provides a highly efficient method for computing gradients.[13] Figure 5 and Table 4 demonstrate how gradient computation scales differently for forward and reverse mode AD and numerical differentiation, looking at the Helmholtz free energy function that has been used in AD literature for benchmarking gradient calculations (Griewank, 1989; Griewank and Walther, 2008; Griewank et al., 2012).

Second-order methods based on Newton's method make use of both the gradient ∇f and the Hessian $\mathbf{H}_f$, working via updates of the form $\Delta\mathbf{w} = -\eta\,\mathbf{H}_f^{-1}\nabla f$ and providing significantly faster convergence (Press et al., 2007). AD provides a way of automatically computing the exact Hessian, enabling succinct and convenient general-purpose implementations.[14] Newton's method converges in fewer iterations, but this comes at the cost of having to compute $\mathbf{H}_f$ in each iteration. In large-scale problems, the Hessian is usually replaced by a numerical approximation using first-order updates from gradient evaluations, giving rise to quasi-Newton methods. A highly popular such method is the BFGS[15] algorithm, together with its limited-memory variant L-BFGS (Dennis and Schnabel, 1996). On the other hand, Hessians arising in large-scale applications are typically sparse. This spar-

13. See `http://DiffSharp.github.io/DiffSharp/examples-gradientdescent.html` for an example of a general-purpose AD-based gradient descent routine using DiffSharp.
14. See `http://DiffSharp.github.io/DiffSharp/examples-newtonsmethod.html` for an implementation of Newton's method with the full Hessian.
15. After Broyden–Fletcher–Goldfarb–Shanno, who independently discovered the method in the 1970s.

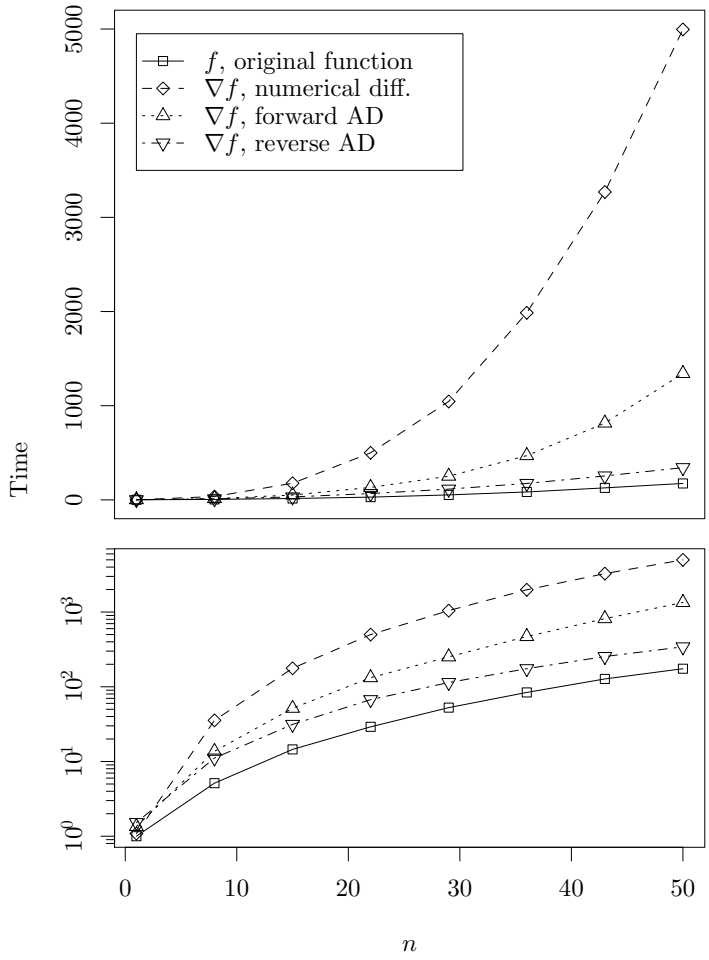

Figure 5: Evaluation time of the Helmholtz free energy function of a mixed fluid, based on the Peng-Robinson equation of state (Peng and Robinson, 1976), $f(\mathbf{x}) = RT \sum_{i=0}^{n} \log \frac{x_i}{1-\mathbf{b^T x}} - \frac{\mathbf{x^T A x}}{\sqrt{8}\mathbf{b^T x}} \log \frac{1+(1+\sqrt{2})\mathbf{b^T x}}{1+(1-\sqrt{2})\mathbf{b^T x}}$, 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 ∇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 $n = 1$. The lower plot uses logarithmic scale for illustrating the behavior for small n. Numerical results are given in Table 4. (Code: `http://DiffSharp.github.io/DiffSharp/misc/Benchmarks-h-grad-v0.5.7.fsx`)

sity 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).

In many cases one does not need the full Hessian but only a Hessian–vector product **Hv**, which can be computed efficiently using a reverse-on-forward configuration of AD by applying the reverse mode to take the gradient of code produced by the forward mode.[16] Given the function $f : \mathbb{R}^n \to \mathbb{R}$, the evaluation point $\mathbf{x}$, and the vector $\mathbf{v}$, one can accomplish this by first computing the directional derivative $\nabla f \cdot \mathbf{v}$ through the forward mode via setting $\dot{\mathbf{x}} = \mathbf{v}$ and then applying the reverse mode on this result to get $\nabla^2 f \cdot \mathbf{v} = \mathbf{H}_f \mathbf{v}$ (Pearlmutter, 1994). This computes $\mathbf{Hv}$ with $O(n)$ complexity, even though $\mathbf{H}$ is a $n \times n$ matrix. Availability of robust AD tools may make more sophisticated optimization methods applicable to large-scale machine-learning problems. For instance, when fast stochastic Hessian–vector products are available, these can be used as the basis of stochastic Newton's methods (Agarwal et al., 2016), which have the potential to endow stochastic optimization with quadratic convergence.

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

4.2 Neural Networks, Deep Learning, Differentiable Programming

Training of a neural network is an optimization problem with respect to its set of weights, which can in principle be addressed by using any method ranging from evolutionary algorithms (Such et al., 2017) to gradient-based methods such as BFGS (Apostolopoulou et al., 2009) or the mainstay stochastic gradient descent (Bottou, 2010) and its many variants (Kingma and Ba, 2015; Tieleman and Hinton, 2012; Duchi et al., 2011). As we have seen, the backpropagation algorithm is only a special case of AD: by applying reverse mode AD to an objective function evaluating a network's error as a function of its weights, we can readily compute the partial derivatives needed for performing weight updates.[17]

The LUSH system (Bottou and LeCun, 2002), and its predecessor SN (Bottou and LeCun, 1988), were the first production systems that targeted efficient neural network simulation while incorporating both a general-purpose programming language and AD. Modern deep learning frameworks provide differentiation capability in one way or another, but the underlying mechanism is not always made clear and confusion abounds regarding the use of the terms "autodiff", "automatic differentiation", and "symbolic differentiation", which

16. Christianson (2012) demonstrates that the second derivative can be computed with the same arithmetic operation sequence using forward-on-reverse, reverse-on-forward, and reverse-on-reverse. The taping overheads of these methods may differ in implementation-dependent ways.
17. See `http://DiffSharp.github.io/DiffSharp/examples-neuralnetworks.html` for an implementation of backpropagation with reverse mode AD.

are sometimes even used interchangeably. In mainstream frameworks including Theano[18] (Bastien et al., 2012), TensorFlow (Abadi et al., 2016), Caffe (Jia et al., 2014), and CNTK (Seide and Agarwal, 2016) the user first constructs a model as a computational graph using a domain-specific mini language, which then gets interpreted by the framework during execution. This approach has the advantage of enabling optimizations of the computational graph structure (e.g., as in Theano), but the disadvantages of having limited and unintuitive control flow and being difficult to debug. In contrast, the lineage of recent frameworks led by autograd (Maclaurin, 2016), Chainer (Tokui et al., 2015), and PyTorch (Paszke et al., 2017) provide truly general-purpose reverse mode AD of the type we outline in Section 3, where the user directly uses the host programming language to define the model as a regular program of the forward computation. This eliminates the need for an interpreter, allows arbitrary control flow statements, and makes debugging simple and intuitive.

Simultaneously with the ongoing adoption of general-purpose AD in machine learning, we are witnessing a modeling-centric terminology emerge within the deep learning community. The terms *define-and-run* and *static computational graph* refer to Theano-like systems where a model is constructed, before execution, as a computational graph structure, which later gets executed with different inputs while remaining fixed. In contrast, the terms *define-by-run* and *dynamic computational graph* refer to the general-purpose AD capability available in newer PyTorch-like systems where a model is a regular program in the host programming language, whose execution dynamically constructs a computational graph on-the-fly that can freely change in each iteration.[19]

Differentiable programming[20] is another emerging term referring to the realization that deep learning models are essentially differentiable program templates of potential solutions to a problem. These templates are constructed as differentiable directed graphs assembled from functional blocks, and their parameters are learned using gradient-based optimization of an objective describing the task. Expressed in this paradigm, neural networks are just a class of parameterized differentiable programs composed of building blocks such as feed-forward, convolutional, and recurrent elements. We are increasingly seeing these traditional building blocks freely composed in arbitrary algorithmic structures using control flow, as well as the introduction of novel differentiable architectures such as the neural Turing machine (Graves et al., 2014), a range of controller–interface abstractions (Graves et al., 2016; Zaremba et al., 2016; Joulin and Mikolov, 2015; Sukhbaatar et al., 2015), and differentiable versions of data structures such as stacks, queues, deques (Grefenstette et al., 2015). Availability of general-purpose AD greatly simplifies the implementation of such architectures by enabling their expression as regular programs that rely on the differentiation infrastructure. Although the differentiable programming perspective on deep learning is new, we note that

18. Theano is a computational graph optimizer and compiler with GPU support and it currently handles derivatives in a highly optimized form of symbolic differentiation. The result can be interpreted as a hybrid of symbolic differentiation and reverse mode AD, but Theano does not use the general-purpose reverse accumulation as we describe in this paper. (Personal communication with the authors.)
19. Note that the terms "static" and "dynamic" here are used in the sense of having a fixed versus non-fixed computational graph topology and not in the sense of data flow architectures.
20. A term advocated by Christopher Olah (`http://colah.github.io/posts/2015-09-NN-Types-FP/`), David Dalrymple (`https://www.edge.org/response-detail/26794`), and Yann LeCun (`https://www.facebook.com/yann.lecun/posts/10155003011462143`) from a deep learning point of view. Note the difference from *differential* dynamic programming (Mayne and Jacobson, 1970) in optimal control.

programming with differentiable functions and having differentiation as a language infrastructure has been the main research subject of the AD community for many decades and realized in a wide range of systems and languages as we shall see in Section 5.

There are instances in neural network literature—albeit few—where explicit reference has been 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 the authors 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. Another example is given for continuous time recurrent neural networks (CTRNN) by Al Seyab and Cao (2008), where the authors apply AD for the training of CTRNNs predicting dynamic behavior of nonlinear processes in real time and report significantly reduced training time compared with other methods.

4.3 Computer Vision

Since the influential work by Krizhevsky et al. (2012), computer vision has been dominated by deep learning, specifically, variations of convolutional neural networks (LeCun et al., 1998). These models are trained end-to-end, meaning that a mapping from raw input data to corresponding outputs is learned, automatically discovering the representations needed for feature detection in a process called representation learning (Bengio et al., 2013).

Besides deep learning, an interesting area where AD can be applied to computer vision problems is inverse graphics (Horn, 1977; Hinton and Ghahramani, 1997)—or analysis-by-synthesis (Yildirim et al., 2015)—where vision is seen as the inference of parameters for a generative model of a scene. Using gradient-based optimization in inverse graphics requires propagating derivatives through whole image synthesis pipelines including the renderer. Eslami et al. (2016) use numerical differentiation for this purpose. Loper and Black (2014) implement the Open Differentiable Renderer (OpenDR), which is a scene renderer that also supplies derivatives of the image pixels with respect to scene parameters, and demonstrate it in the task of fitting an articulated and deformable 3D model of the human body to image and range data from a Kinect device. Similarly, Kulkarni et al. (2015) implement a differentiable approximate renderer for the task of inference in probabilistic programs describing scenes.

Srajer et al. (2016) investigate the use of AD for three tasks in computer vision and machine learning, namely bundle adjustment (Triggs et al., 1999), Gaussian mixture model fitting, and hand tracking (Taylor et al., 2014), and provide a comprehensive benchmark of various AD tools for the computation of derivatives in these tasks.

Pock et al. (2007) make use of AD in addressing 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, Figure 5 and Table 4).

Barrett and Siskind (2013) present a use of general-purpose AD for the task of video event detection using hidden Markov models (HMMs) and Dalal and Triggs (2005) object detectors, performing training on a corpus of pre-tracked video using an adaptive step size gradient descent with reverse mode AD. Initially implemented with the R6RS-AD package[21] which provides forward and reverse mode AD in Scheme, the resulting gradient code was later ported to C and highly optimized.[22]

4.4 Natural Language Processing

Natural language processing (NLP) constitutes one of the areas where rapid progress is being made by applying deep learning techniques (Goldberg, 2016), with applications in tasks including machine translation (Bahdanau et al., 2014), language modeling (Mikolov et al., 2010), dependency parsing (Chen and Manning, 2014), and question answering (Kumar et al., 2016). Besides deep learning approaches, statistical models in NLP are commonly trained using general purpose or specialized gradient-based 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 uses 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.[23]

4.5 Probabilistic Modeling and Inference

Inference in probabilistic models can be static, such as compiling a given model to Bayesian networks and using algorithms such as belief propagation for inference; or they can be dynamic, executing a model forward many times and computing statistics on observed values to infer posterior distributions. Markov chain Monte Carlo (MCMC) (Neal, 1993) methods are often used for dynamic inference, such as the Metropolis–Hastings algorithm based on random sampling (Chib and Greenberg, 1995). Meyer et al. (2003) give an example of how AD can be used to speed up Bayesian posterior inference in MCMC, with an application in stochastic volatility. Amortized inference (Gershman and Goodman, 2014; Stuhlmüller et al., 2013) techniques based on deep learning (Le et al., 2017; Ritchie et al., 2016) work by training neural networks for performing approximate inference in generative models defined as probabilistic programs (Gordon et al., 2014).

When model parameters are continuous, the Hamiltonian—or, hybrid—Monte Carlo (HMC) algorithm provides improved convergence characteristics avoiding the slow explo-

21. `https://github.com/qobi/R6RS-AD`
22. Personal communication.
23. 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`

ration of random sampling, by simulating Hamiltonian dynamics through auxiliary "momentum variables" (Duane et al., 1987). The advantages of HMC come at the cost of requiring gradient evaluations of complicated probability models. AD is highly suitable here for complementing probabilistic modeling, because it relieves the user from the manual derivation of gradients for each model.[24] For instance, the probabilistic programming language Stan (Carpenter et al., 2016) 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 (Carpenter et al., 2015). Similarly, Wingate et al. (2011) demonstrate the use of AD as a non-standard interpretation of probabilistic programs enabling efficient inference algorithms. Kucukelbir et al. (2017) present an AD-based method for deriving variational inference (VI) algorithms.

PyMC3 (Salvatier et al., 2016) allows fitting of Bayesian models using MCMC and VI, for which it uses gradients supplied by Theano. Edward (Tran et al., 2016) is a library for deep probabilistic modeling, inference, and criticism (Tran et al., 2017) that supports VI using TensorFlow. Availability of general-purpose AD in this area has enabled new libraries such as Pyro[25] and ProbTorch (Siddharth et al., 2017) for deep *universal* probabilistic programming with support for recursion and control flow, relying, in both instances, on VI using gradients supplied by PyTorch's reverse mode AD infrastructure.

When working with probabilistic models, one often needs to backpropagate derivatives through sampling operations of random variables in order to achieve stochastic optimization of model parameters. The score-function estimator, or REINFORCE (Williams, 1992), method provides a generally applicable unbiased gradient estimate, albeit with high variance. When working with continuous random variables, one can substitute a random variable by a deterministic and differentiable transformation of a simpler random variable, a method known as the "reparameterization trick" (Williams, 1992; Kingma and Welling, 2014; Rezende et al., 2014). For discrete variables, the REBAR (Tucker et al., 2017) method provides a lower-variance unbiased gradient estimator by using continuous relaxation. A generalization of REBAR called RELAX (Grathwohl et al., 2017) works by learning a free-form control variate parameterized by a neural network and is applicable in both discrete and continuous settings.

5. Implementations

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 guarantees that the amount of arithmetic goes up by no more than a small constant factor (Griewank and Walther, 2008). On the other hand, managing this arithmetic can introduce a significant overhead if done carelessly. For instance, naïvely allocating data structures for holding dual numbers will involve memory access and allocation for every arithmetic opera-

24. See `http://diffsharp.github.io/DiffSharp/examples-hamiltonianmontecarlo.html` for an implementation of HMC with reverse mode AD.

25. `http://pyro.ai/`

tion, which are usually more expensive than arithmetic operations on modern computers.[26] 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.[27]

Another major issue is the risk of hitting a class of bugs called "perturbation confusion" (Siskind and Pearlmutter, 2005; Manzyuk et al., 2012). This essentially means that if two ongoing differentiations affect the same piece of code, the two formal epsilons they introduce (Section 3.1.1) need to be kept distinct. It is very easy to have bugs—particularly in 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 compute derivatives.

Translation of mathematics into computer code often requires attention to numeric issues. For instance, the mathematical expressions $\log(1+x)$ or $\sqrt{x^2+y^2+z^2}$ or $\tan^{-1}(y/x)$ should not be naïvely translated, but rather expressed as `log1p(x)`, `hypot(x,hypot(y,z))`, and `atan2(y,x)`. In machine learning, the most prominent example of this is probably the so-called log-sum-exp trick to improve the numerics of calculations of the form $\log \sum_i \exp x_i$. AD is not immune to such numeric considerations. For example, code calculating $E = \sum_i E_i$, processed by AD, will calculate $\nabla_w E = \sum_i \nabla_w E_i$. If the system is seeking a local minimum of E then $\nabla_w E = \sum_i \nabla_w E_i \to_t 0$, and naïvely adding a set of large numbers whose sum is near zero is numerically fraught. This is to say that AD is not immune to the perils of floating point arithmetic, and can sometimes introduce numeric issues which were not present in the primal calculation. Issues of numeric analysis are outside our present scope, but there is a robust literature on the numerics of AD (e.g., Griewank et al. (2012)) involving using subgradients to allow optimization to proceed despite non-differentiability of the objective, appropriate subgradients and approximations for functions like $|\cdot|$ and $\|\cdot\|_2$ and $\sqrt{\cdot}$ near zero, and a spate of related issues.

One should also be cautious about approximated functions and AD (Sirkes and Tziperman, 1997). In this case, if one has 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 the procedure was approximating. For instance, 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.[28] 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 mathe-

26. The implementation of forward mode in Julia (Revels et al., 2016b) attempts to avoid this, and some current compilers can avoid this expense by unboxing dual numbers (Leroy, 1997; Jones et al., 1993b; Jones and Launchbury, 1991; Siskind and Pearlmutter, 2016). This method is also used to reduce the memory-access overhead in the implementations of forward mode in Stalingrad and the Haskell *ad* library.
27. Flow analysis (Shivers, 1991) and/or partial evaluation (Jones et al., 1993a), together with tag stripping (Appel, 1989; Peterson, 1989), can remove this method dispatch. These, together with unboxing, can often make it possible to completely eliminate the memory access, memory allocation, memory reclamation, and method dispatch overhead of dual numbers (Siskind and Pearlmutter, 2016).
28. In modern systems this is not an issue, because e^x is a primitive implemented in hardware.

matical function can only be computed approximately but has a well-defined mathematical derivative.

We note that there are similarities as well as differences between machine learning workloads and those studied in the traditional AD literature (Baydin et al., 2016b). Deep learning systems are generally compute-bound and spend a considerable amount of computation time in highly-optimized numerical kernels for matrix operations (Hadjis et al., 2015; Chetlur et al., 2014). This is a situation which is arguably amenable to operator-overloading-based AD implementations on high-level operations, as is commonly found in current machine learning frameworks. In contrast, numerical simulation workloads in traditional AD applications can be bandwidth-bound, making source code transformation and compiler optimization approaches more relevant. Another difference worth noting is that whereas high numerical precision is desirable in traditional application domains of AD such as computational fluid dynamics (Cohen and Molemaker, 2009), in deep learning lower-precision is sufficient and even desirable in improving computational efficiency, thanks to the error resiliency of neural networks (Gupta et al., 2015; Courbariaux et al., 2015).

There are instances in recent literature where implementation-related experience from the AD field has been put to use in machine learning settings. One particular area of recent interest is implicit and iterative AD techniques (Griewank and Walther, 2008), which has found use in work incorporating constrained optimization within deep learning (Amos and Kolter, 2017) and probabilistic graphical models and neural networks (Johnson et al., 2016). Another example is checkpointing strategies (Dauvergne and Hascoët, 2006; Siskind and Pearlmutter, 2017), which allow balancing of application-specific trade-offs between time and space complexities of reverse mode AD by not storing the full tape of intermediate variables in memory and reconstructing these as needed by re-running parts of the forward computation from intermediate checkpoints. This is highly relevant in deep learning workloads running on GPUs with limited memory budgets. A recent example in this area is the work by Gruslys et al. (2016), where the authors construct a checkpointing variety of the backpropagation through time (BPTT) algorithm for recurrent neural networks and demonstrate it saving up to 95% memory usage at the cost of a 33% increase in computation time in one instance.

In Table 5 we present a review of notable general-purpose AD implementations.[29] 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 the following part of this section.

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 when writing the code.

The approach has been utilized since the early days of AD, with prototypical examples being the WCOMP and UCOMP packages of Lawson (1971), the APL package of Neidinger

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

Table 5: Survey of AD implementations. Tools developed primarily for machine learning are highlighted in bold.

Language	Tool	Type	Mode	Institution / Project	Reference	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.mcs.anl.gov/research/projects/adic/
	ADOL-C	OO	F, R	Computational Infrastructure for Operations Research	Walther and Griewank (2012)	https://projects.coin-or.org/ADOL-C
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)	
C#	AutoDiff	LIB	R	George Mason Univ., Dept. of Computer Science	Shtof et al. (2013)	http://autodiff.codeplex.com/
F#, C#	**DiffSharp**	OO	F, R	Maynooth University, Microsoft Research Cambridge	Baydin et al. (2016a)	http://diffsharp.github.io
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	COSY	INT	F	Michigan State Univ., Biomedical and Physical Sci.	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	ADiJaC	ST	F, R	University Politehnica of Bucharest	Slusanschi and Dumitrel (2016)	http://adijac.cs.pub.ro
	Deriva	LIB	R	Java & Clojure library		https://github.com/lambder/Deriva
Julia	JuliaDiff	OO	F, R	Julia packages	Revels et al. (2016a)	http://www.juliadiff.org/
Lua	**torch-autograd**	OO	R	Twitter Cortex		https://github.com/twitter/torch-autograd
MATLAB	ADiMat	ST	F, R	Technical University of Darmstadt, Scientific Comp.	Willkomm and Vehreschild (2013)	http://adimat.sc.informatik.tu-darmstadt.de/
	INTLab	OO	F	Hamburg Univ. of Technology, Inst. for Reliable Comp.	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
	autograd	OO	F, R	Harvard Intelligent Probabilistic Systems Group	Maclaurin (2016)	https://github.com/HIPS/autograd
	Chainer	OO	R	Preferred Networks	Tokui et al. (2015)	https://chainer.org/
	PyTorch	OO	R	PyTorch core team	Paszke et al. (2017)	http://pytorch.org/
	Tangent	ST	F, R	Google Brain	van Merriënboer et al. (2017)	https://github.com/google/tangent
Scheme	R6RS-AD	OO	F, R	Purdue Univ., School of Electrical and Computer Eng.		https://github.com/qobi/R6RS-AD
	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
	Stalingrad	COM	F, R	Purdue Univ., School of Electrical and Computer Eng.	Pearlmutter and Siskind (2008)	http://www.bcl.hamilton.ie/~qobi/stalingrad/

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

(1989), and the work by Hinkins (1994). Likewise, Rich and Hill (1992) formulate their implementation of AD in MATLAB using elemental methods.

Elemental libraries still constitute the simplest strategy to implement AD for languages without operator overloading.

5.2 Compilers and Source Code Transformation

These implementations provide extensions to programming languages that automate the decomposition of algorithms into AD-enabled elementary operations. They are typically executed as preprocessors[30] 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 (Bischof et al., 1996), given a 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 code 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.[31]

In addition to language extensions through source code transformation, there are implementations introducing new languages with tightly integrated AD capabilities 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 objectives and constraints to be expressed 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; Siskind and Pearlmutter, 2008a), working on the Scheme-based AD-aware VLAD language, also falls under this category. The newer DVL compiler[32] is based on Stalingrad and uses a reimplementation of portions of the VLAD language.

Motivated by machine learning applications, the Tangent library (van Merriënboer et al., 2017) implements AD using source code transformation, and accepts numeric functions written in a syntactic subset of Python and Numpy.

30. Preprocessors transform program source code before it is given as an input to a compiler.
31. `http://www-tapenade.inria.fr:8080/tapenade/index.jsp`
32. `https://github.com/axch/dysvunctional-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 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 records arithmetic operations on variables in tape data structures, 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 operator overloading. For Python, the *ad* package[33] uses operator overloading to compute first- and second-order derivatives, while the newer autograd package[34] provides forward and reverse mode AD with support for higher-order derivatives.

For functional languages, examples include R6RS-AD[35] and the AD routines within the Scmutils library[36] for Scheme, the *ad* library[37] for Haskell, and DiffSharp[38] for F# and C#.

6. Conclusions

Backpropagation and gradient-based optimization are behind virtually all recent successes in machine learning, yielding state-of-the-art results in computer vision, speech recognition and synthesis, and machine translation. We expect these techniques to remain at the core of machine learning for the foreseeable future. Research in the field involves a rapid prototyping and development cycle for testing new models and ideas, using a collection of increasingly higher-quality machine learning frameworks. These frameworks are in the process of transition from coarse-grained (module level) backpropagation towards fine-grained, general-purpose AD, allowing models to be implemented as regular programs in general-purpose programming languages with differentiation as an integral part of the infrastructure. We strongly believe that general-purpose AD is the future of gradient-based machine learning and we expect it to become an indispensable tool in the machine learning toolbox.

It is an exciting time for working at the intersection of AD and machine learning, and there are many opportunities for bringing advanced techniques and expertise from AD literature to bear on machine learning problems. Techniques that have been developed by the AD community such as tape reduction and elimination (Naumann, 2004), fixed-point iterations (Christianson, 1994), utilizing sparsity by matrix coloring (Gebremedhin et al., 2009, 2013), and reverse AD checkpointing (Dauvergne and Hascoët, 2006) are just a few examples that can find potential use in machine learning for increasing performance, improving convergence of optimization, using hardware more efficiently, and even enabling

33. http://pythonhosted.org/ad/
34. https://github.com/HIPS/autograd
35. https://github.com/NUIM-BCL/R6RS-AD
36. http://groups.csail.mit.edu/mac/users/gjs/6946/refman.txt
37. http://hackage.haskell.org/package/ad
38. http://diffsharp.github.io

new types of machine learning models to be implemented. Similarly, exciting new AD modes like direct propagation of the inverse Jacobian (Srinivasan and Todorov, 2015) have emerged from the machine learning community, but have yet to be examined and formalized by the AD community.

An important direction for future work is to make use of nested AD techniques in machine learning, allowing differentiation to be nested arbitrarily deep with referential transparency (Siskind and Pearlmutter, 2008b; Pearlmutter and Siskind, 2008). Nested AD is highly relevant in hyperparameter optimization as it can effortlessly provide exact hypergradients, that is, derivatives of a training objective with respect to the hyperparameters of an optimization routine (Maclaurin et al., 2015; Baydin et al., 2018). Potential applications include Bayesian model selection (Rasmussen and Williams, 2006) and gradient-based tuning of Hamiltonian Monte Carlo step sizes and mass matrices (Salimans et al., 2015). Besides hyperparameters, models internally using higher-order derivatives constitute a straightforward usage case for nested AD. The Riemannian manifold Langevin and Hamiltonian Monte Carlo methods (Girolami and Calderhead, 2011) use higher-order derivative information to more closely track the information geometry of the sampled distribution for faster convergence and exploration. In neural networks, it is very natural to use nested derivatives in defining objective functions that take input transformations into account, such as the Tangent Prop method (Simard et al., 1998) for imposing invariance under a set of chosen transformations.

Acknowledgments

We thank the anonymous reviewers whose comments helped improve this manuscript. This work was supported, in part, by Science Foundation Ireland grant 09/IN.1/I2637, by the Army Research Laboratory, accomplished under Cooperative Agreement Number W911NF-10-2-0060, by the National Science Foundation under Grants 1522954-IIS and 1734938-IIS, and by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior/Interior Business Center (DOI/IBC) contract number D17PC00341. Any opinions, findings, views, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, official policies, or endorsements, either expressed or implied, of the sponsors. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

D. S. Adamson and C. W. Winant. 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*, pages 231–40, 1969.

Naman Agarwal, Brian Bullins, and Elad Hazan. Second order stochastic optimization in linear time. Technical Report arXiv:1602.03943, arXiv preprint, 2016.

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. doi: 10.1016/j.jprocont.2007.10.012.

Brandon Amos and J Zico Kolter. OptNet: Differentiable optimization as a layer in neural networks. *arXiv preprint arXiv:1703.00443*, 2017.

Marianna S. Apostolopoulou, Dimitris G. Sotiropoulos, Ioannis E. Livieris, and Panagiotis Pintelas. A memoryless BFGS neural network training algorithm. In *7th IEEE International Conference on Industrial Informatics, INDIN 2009*, pages 216–221, June 2009. doi: 10.1109/INDIN.2009.5195806.

Andrew W Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2(2): 153–162, 1989.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Daniel Paul Barrett and Jeffrey Mark Siskind. Felzenszwalb-Baum-Welch: Event detection by changing appearance. *arXiv preprint arXiv:1306.4746*, 2013.

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

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

Atılım Güneş Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Diffsharp: An AD library for .NET languages. In *7th International Conference on Algorithmic Differentiation, Christ Church Oxford, UK, September 12–15, 2016*, 2016a. Also arXiv:1611.03423.

Atılım Güneş Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Tricks from deep learning. In *7th International Conference on Algorithmic Differentiation, Christ Church Oxford, UK, September 12–15, 2016*, 2016b. Also arXiv:1611.03777.

Atılım Güneş Baydin, Robert Cornish, David Martínez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. In *Sixth International Conference on Learning Representations (ICLR), Vancouver, Canada, April 30–May 3, 2018*, 2018.

L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. Programs for automatic differentiation for the machine BESM (in Russian). Technical report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, 1959.

Bradley M. Bell and James V. Burke. Algorithmic differentiation of implicit functions and optimal values. In C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*, pages 67–77. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-68942-3_7.

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

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

Charles W. Bert and Moinuddin Malik. Differential quadrature method in computational mechanics: A review. *Applied Mechanics Reviews*, 49, 1996. doi: 10.1115/1.3101882.

Martin Berz, Kyoko Makino, Khodr Shamseddine, Georg H. Hoffstätter, and Weishi Wan. COSY INFINITY and its applications in nonlinear dynamics. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 363–5. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1996.

Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *Computational Science Engineering, IEEE*, 3(3):18–32, 1996. doi: 10.1109/99.537089.

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

Christian H. Bischof, H. Martin Bücker, and Bruno Lang. Automatic differentiation for computational finance. In E. J. Kontoghiorghes, B. Rustem, and S. Siokos, editors, *Computational Methods in Decision-Making, Economics and Finance*, volume 74 of *Applied Optimization*, pages 297–310. Springer US, 2002. doi: 10.1007/978-1-4757-3613-7_15.

Christian H. Bischof, H. Martin Bücker, Arno Rasch, Emil Slusanschi, and Bruno Lang. Automatic differentiation of the general-purpose computational fluid dynamics package FLUENT. *Journal of Fluids Engineering*, 129(5):652–8, 2006. doi: 10.1115/1.2720475.

Christian H. Bischof, Paul D. Hovland, and Boyana Norris. On the implementation of automatic differentiation tools. *Higher-Order and Symbolic Computation*, 21(3):311–31, 2008. doi: 10.1007/s10990-008-9034-4.

V. G. Boltyanskii, R. V. Gamkrelidze, and L. S. Pontryagin. The theory of optimal processes I: The maximum principle. *Izvest. Akad. Nauk S.S.S.R. Ser. Mat.*, 24:3–42, 1960.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

Léon Bottou and Yann LeCun. SN: A simulator for connectionist models. In *Proceedings of NeuroNimes 88*, pages 371–382, Nimes, France, 1988. URL `http://leon.bottou.org/papers/bottou-lecun-88`.

Léon Bottou and Yann LeCun. Lush reference manual, 2002. URL `http://lush.sourceforge.net/doc.html`.

Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

Léon Bottou. Online learning and stochastic approximations. *On-Line Learning in Neural Networks*, 17:9, 1998.

Claude Brezinski and M. Redivo Zaglia. *Extrapolation Methods: Theory and Practice*. North-Holland, 1991.

A. E. Bryson and W. F. Denham. A steepest ascent method for solving optimum programming problems. *Journal of Applied Mechanics*, 29(2):247, 1962. doi: 10.1115/1.3640537.

Arthur E. Bryson and Yu-Chi Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. Blaisdell, Waltham, MA, 1969.

Rirchard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, 2001.

Luca Capriotti. Fast Greeks by algorithmic differentiation. *Journal of Computational Finance*, 14(3):3, 2011.

Gregory R. Carmichael and Adrian Sandu. Sensitivity analysis for atmospheric chemistry models via automatic differentiation. *Atmospheric Environment*, 31(3):475–89, 1997.

Bob Carpenter, Matthew D Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv preprint arXiv:1509.07164*, 2015.

Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 20:1–37, 2016.

Daniele Casanova, Robin S. Sharp, Mark Final, Bruce Christianson, and Pat Symonds. Application of automatic diffentiation to race car performance optimisation. In George Corliss, Christèle Faure, Andreas Griewank, Lauren Hascoët, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms*, pages 117–124. Springer-Verlag New York, Inc., New York, NY, USA, 2002. ISBN 0-387-95305-1.

Isabelle Charpentier and Mohammed Ghemires. Efficient adjoint derivatives: Application to the meteorological model Meso-NH. *Optimization Methods and Software*, 13(1):35–63, 2000.

Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, 2014.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Siddhartha Chib and Edward Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, 1995. doi: 10.1080/00031305.1995.10476177.

Bruce Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3(4):311–326, 1994.

Bruce Christianson. A Leibniz notation for automatic differentiation. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 1–9. Springer, Berlin, 2012. ISBN 978-3-540-68935-5. doi: 10.1007/978-3-642-30023-3_1.

William K. Clifford. Preliminary sketch of bi-quaternions. *Proceedings of the London Mathematical Society*, 4:381–95, 1873.

J Cohen and M Jeroen Molemaker. A fast double precision cfd code using cuda. *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 414–429, 2009.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

George F. Corliss. *Application of differentiation arithmetic*, volume 19 of *Perspectives in Computing*, pages 127–48. Academic Press, Boston, 1988.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

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

Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 566–73, Dauvergne, 2006. Springer Berlin.

John E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

L. C. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 114–125. SIAM, Philadelphia, PA, 1991.

Simon Duane, Anthony D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid Monte Carlo. *Physics Letters B*, 195(2):216–222, 1987.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

Ulf Ekström, Lucas Visscher, Radovan Bast, Andreas J. Thorvaldsen, and Kenneth Ruud. Arbitrary-order density functional response theory from automatic differentiation. *Journal of Chemical Theory and Computation*, 6:1971–80, 2010. doi: 10.1021/ct100117s.

Jerry Eriksson, Mårten Gulliksson, Per Lindström, and Per Åke Wedin. Regularization tools for training large feed-forward neural networks using automatic differentiation. *Optimization Methods and Software*, 10(1):49–69, 1998. doi: 10.1080/10556789808805701.

S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Koray Kavukcuoglu, and Geoffrey E. Hinton. Attend, infer, repeat: Fast scene understanding with generative models. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3225–3233. Curran Associates, Inc., 2016.

Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. Efficient, feature-based, conditional random field parsing. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL 2008)*, pages 959–67, 2008.

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

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

Shaun A. Forth and Trevor P. Evans. Aerofoil optimisation via AD of a multigrid cell-vertex Euler flow solver. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, pages 153–160. Springer New York, New York, NY, 2002. ISBN 978-1-4613-0075-5. doi: 10.1007/978-1-4613-0075-5_17.

Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.

David M. Gay. Automatically finding and exploiting partially separable structure in nonlinear programming problems. Technical report, Bell Laboratories, Murray Hill, NJ, 1996.

Assefaw H. Gebremedhin, Arijit Tarafdar, Alex Pothen, and Andrea Walther. Efficient computation of sparse Hessians using coloring and automatic differentiation. *INFORMS Journal on Computing*, 21(2):209–23, 2009. doi: 10.1287/ijoc.1080.0286.

Assefaw H Gebremedhin, Duc Nguyen, Md Mostofa Ali Patwary, and Alex Pothen. ColPack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software (TOMS)*, 40(1):1, 2013.

Samuel Gershman and Noah Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, number 36, 2014.

Ralf Giering and Thomas Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24:437–74, 1998. doi: 10.1145/293686.293695.

Kevin Gimpel, Dipanjan Das, and Noah A. Smith. Distributed asynchronous online learning for natural language processing. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, CoNLL '10, pages 213–222, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

Mark Girolami and Be Calderhead. Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(2):123–214, 2011.

Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

Johannes Grabmeier and Erich Kaltofen. *Computer Algebra Handbook: Foundations, Applications, Systems*. Springer, 2003.

Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz. Automatic differentiation for GPU-accelerated 2D/3D registration. In C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*, pages 259–269. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-68942-3_23.

Will Grathwohl, Dami Choi, Yuhuai Wu, Geoff Roeder, and David Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. *arXiv preprint arXiv:1711.00123*, 2017.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12: 321–98, 2003. doi: 10.1017/S0962492902000132.

Andreas Griewank. Who invented the reverse mode of differentiation? *Documenta Mathematica*, Extra Volume ISMP:389–400, 2012.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, 2008. doi: 10.1137/1.9780898717761.

Andreas Griewank, Kshitij Kulshreshtha, and Andrea Walther. On the numerical stability of algorithmic differentiation. *Computing*, 94(2-4):125–149, 2012.

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1737–1746, 2015.

Gundolf Haase, Ulrich Langer, Ewald Lindner, and Wolfram Mühlhuber. Optimal sizing of industrial structural mechanics problems using AD. In *Automatic Differentiation of Algorithms*, pages 181–188. Springer, 2002.

Stefan Hadjis, Firas Abuzaid, Ce Zhang, and Christopher Ré. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, page 2. ACM, 2015.

William Rowan Hamilton. Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time. *Transactions of the Royal Irish Academy*, 17:293–422, 1837.

Laurent Hascoët and Valérie Pascual. The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3), 2013. doi: 10.1145/2450153.2450158.

Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *International Joint Conference on Neural Networks, IJCNN 1989*, pages 593–605. IEEE, 1989.

Ruth L. Hinkins. Parallel computation of automatic differentiation applied to magnetic field calculations. Technical report, Lawrence Berkeley Lab., CA, 1994.

Geoffrey E. Hinton and Zoubin Ghahramani. Generative models for discovering sparse distributed representations. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 352(1358):1177–1190, 1997.

Matthew D. Hoffman and Andrew Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15: 1351–1381, 2014.

Berthold K. P. Horn. Understanding image intensities. *Artificial Intelligence*, 8:201–231, 1977.

Jim E. Horwedel, Brian A. Worley, E. M. Oblow, and F. G. Pin. GRESS version 1.0 user's manual. Technical Memorandum ORNL/TM 10835, Martin Marietta Energy Systems, Inc., Oak Ridge National Laboratory, Oak Ridge, 1988.

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

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

Matthew Johnson, David K Duvenaud, Alex Wiltschko, Ryan P Adams, and Sandeep R Datta. Composing graphical models with neural networks for structured representations and fast inference. In *Advances in Neural Information Processing Systems*, pages 2946–2954, 2016.

Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993a.

Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993b.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems*, pages 190–198, 2015.

David W. Juedes. A taxonomy of automatic differentiation tools. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–29. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *The International Conference on Learning Representations (ICLR), San Diego*, 2015.

Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations*, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

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

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic differentiation variational inference. *Journal of Machine Learning Research*, 18(14):1–45, 2017.

Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1378–1387, New York, New York, USA, 20–22 Jun 2016. PMLR.

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

Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

G. W. Leibniz. *Machina arithmetica in qua non additio tantum et subtractio sed et multiplicatio nullo, diviso vero paene nullo animi labore peragantur*. Hannover, 1685.

Xavier Leroy. The effectiveness of type-based unboxing. In *TIC 1997: Workshop Types in Compilation*, 1997.

Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's thesis, University of Helsinki, 1970.

Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

Matthew M. Loper and Michael J. Black. OpenDR: An approximate differentiable renderer. In *European Conference on Computer Vision*, pages 154–169. Springer, 2014.

Dougal Maclaurin. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. PhD thesis, School of Engineering and Applied Sciences, Harvard University, 2016.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R Rush, and Jeffrey Mark Siskind. Confusion of tagged perturbations in forward automatic differentiation of higher-order functions. *arXiv preprint arXiv:1211.4892*, 2012.

David Q. Mayne and David H. Jacobson. *Differential Dynamic Programming*. American Elsevier Pub. Co., New York, 1970.

Vladimir Mazourik. Integration of automatic differentiation into a numerical library for PC's. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–29. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.

Renate Meyer, David A. Fournier, and Andreas Berg. Stochastic volatility: Bayesian computation using automatic differentiation and the extended Kalman filter. *Econometrics Journal*, 6(2):408–420, 2003. doi: 10.1111/1368-423X.t01-1-00116.

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

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

J. D. Müller and P. Cusdin. On the performance of discrete adjoint CFD codes using automatic differentiation. *International Journal for Numerical Methods in Fluids*, 47 (8-9):939–945, 2005. ISSN 1097-0363. doi: 10.1002/fld.885.

Uwe Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming*, 99(3):399–421, 2004.

Uwe Naumann and Jan Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 159–69. Springer, 2005.

Radford M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto, 1993.

Richard D. Neidinger. Automatic differentiation and APL. *College Mathematics Journal*, 20(3):238–51, 1989. doi: 10.2307/2686776.

John F. Nolan. Analytical differentiation on a digital computer. Master's thesis, Massachusetts Institute of Technology, 1953.

J. F. Ostiguy and L. Michelotti. Mxyzptlk: An efficient, native C++ differentiation engine. In *Particle Accelerator Conference (PAC 2007)*, pages 3489–91. IEEE, 2007. doi: 10.1109/PAC.2007.4440468.

David B. Parker. Learning-logic: Casting the cortex of the human brain in silicon. Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, 1985.

Valérie Pascual and Laurent Hascoët. TAPENADE for C. In *Advances in Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, pages 199–210. Springer, 2008. doi: 10.1007/978-3-540-68942-3_18.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, Long Beach, CA, US, December 9, 2017*, 2017.

Barak A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6:147–60, 1994. doi: 10.1162/neco.1994.6.1.147.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):1–36, March 2008. doi: 10.1145/1330017.1330018.

Ding-Yu Peng and Donald B. Robinson. A new two-constant equation of state. *Industrial and Engineering Chemistry Fundamentals*, 15(1):59–64, 1976. doi: 10.1021/i160057a011.

John Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 89–99. ACM, 1989.

F. W. Pfeiffer. Automatic differentiation in PROSE. *SIGNUM Newsletter*, 22(1):2–8, 1987. doi: 10.1145/24680.24681.

Thomas Pock, Michael Pock, and Horst Bischof. Algorithmic differentiation: Application to variational problems in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(7):1180–1193, 2007. doi: 10.1109/TPAMI.2007.1044.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007.

Louise B. Rall. Perspectives on automatic differentiation: Past, present, and future? In M. Bücker, G. Corliss, U. Naumann, P. Hovland, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 1–14. Springer Berlin Heidelberg, 2006.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. MIT Press, 2006.

J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv:1607.07892 [cs.MS]*, 2016a. URL `https://arxiv.org/abs/1607.07892`.

Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv preprint arXiv:1607.07892*, 2016b.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pages 1278–1286, 2014.

Lawrence C. Rich and David R. Hill. Automatic differentiation in MATLAB. *Applied Numerical Mathematics*, 9:33–43, 1992.

Daniel Ritchie, Paul Horsfall, and Noah D Goodman. Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*, 2016.

Elizabeth Rollins. Optimization of neural network feedback control systems using automatic differentiation. Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 2009.

L. I. Rozonoer. L. S. Pontryagin's maximum principle in the theory of optimum systems—Part II. *Automat. i Telemekh.*, 20:1441–1458, 1959.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

Siegfried M. Rump. INTLAB—INTerval LABoratory. In *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. doi: 10.1007/978-94-017-1247-7_7.

Tim Salimans, Diederik Kingma, and Max Welling. Markov chain Monte Carlo and variational inference: Bridging the gap. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1218–1226, 2015.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55, 2016.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351, 2013.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

Nicol N. Schraudolph. Local gain adaptation in stochastic gradient descent. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 569–74, Edinburgh, Scotland, 1999. IEE London. doi: 10.1049/cp:19991170.

Nicol N. Schraudolph and Thore Graepel. Combining conjugate direction methods with stochastic approximation of gradients. In *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, 2003.

Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2945397.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations 2017*, 2017.

Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

Alex Shtof, Alexander Agathos, Yotam Gingold, Ariel Shamir, and Daniel Cohen-Or. Geosemantic snapping for sketch-based modeling. *Computer Graphics Forum*, 32(2):245–53, 2013. doi: 10.1111/cgf.12044.

N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. Learning disentangled representations with semi-supervised deep generative models. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5927–5937. Curran Associates, Inc., 2017.

Patrice Simard, Yann LeCun, John Denker, and Bernard Victorri. Transformation invariance in pattern recognition, tangent distance and tangent propagation. In G. Orr and K. Muller, editors, *Neural Networks: Tricks of the Trade*. Springer, 1998.

Z. Sirkes and E. Tziperman. Finite difference of adjoint or adjoint of finite difference? *Monthly Weather Review*, 125(12):3373–8, 1997. doi: 10.1175/1520-0493(1997)125⟨3373:FDOAOA⟩2.0.CO;2.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In Andrew Butterfield, editor, *Implementation and Application of Functional Languages—17th International Workshop, IFL'05*, pages 1–9, Dublin, Ireland, 2005. Trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60.

Jeffrey Mark Siskind and Barak 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, School of Electrical and Computer Engineering, Purdue University, 2008a.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, 2008b.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Efficient implementation of a higher-order language with built-in AD. In *7th International Conference on Algorithmic Differentiation, Christ Church Oxford, UK, September 12–15, 2016*, 2016. Also arXiv:1611.03416.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, Long Beach, CA, US, December 9, 2017*, 2017. Also arXiv:1708.06799.

Emil I. Slusanschi and Vlad Dumitrel. ADiJaC—Automatic differentiation of Java classfiles. *ACM Transaction on Mathematical Software*, 43(2):9:1–9:33, September 2016. ISSN 0098-3500. doi: 10.1145/2904901.

Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

Suvrit Sra, Sebastian Nowozin, and Stephen J. Wright. *Optimization for Machine Learning*. MIT Press, 2011.

Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in machine learning and computer vision. In *AD2016: The 7th International Conference on Algorithmic Differentiation, Monday 12th–Thursday 15th September 2016, Christ Church Oxford, UK: Programme and Abstracts*, pages 181–184. Society for Industrial and Applied Mathematics (SIAM), 2016.

Akshay Srinivasan and Emanuel Todorov. Graphical Newton. Technical Report arXiv:1508.00952, arXiv preprint, 2015.

Andreas Stuhlmüller, Jacob Taylor, and Noah Goodman. Learning stochastic inverses. In *Advances in Neural Information Processing Systems*, pages 3048–3056, 2013.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, pages 2440–2448, 2015.

Gerald J. Sussman and Jack Wisdom. *Structure and Interpretation of Classical Mechanics*. MIT Press, 2001. doi: 10.1063/1.1457268.

Jonathan Taylor, Richard Stebbing, Varun Ramakrishna, Cem Keskin, Jamie Shotton, Shahram Izadi, Aaron Hertzmann, and Andrew Fitzgibbon. User-specific hand modeling from monocular depth sequences. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 644–651, 2014.

Jeffrey P. Thomas, Earl H. Dowell, and Kenneth C. Hall. Using automatic differentiation to create a nonlinear reduced order model of a computational fluid dynamic solver. *AIAA Paper*, 7115:2006, 2006.

T. Tieleman and G. Hinton. Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4 (2), 2012.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment—a modern synthesis. In *International Workshop on Vision Algorithms*, pages 298–372. Springer, 1999.

George Tucker, Andriy Mnih, Chris J. Maddison, John Lawson, and Jascha Sohl-Dickstein. REBAR: Low-variance, unbiased gradient estimates for discrete latent variable models. In *Advances in Neural Information Processing Systems*, pages 2624–2633, 2017.

Bart van Merriënboer, Alexander B. Wiltschko, and Dan Moldovan. Tangent: Automatic differentiation using source code transformation in Python. *arXiv preprint arXiv:1711.02712*, 2017.

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

S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark W. Schmidt, and Kevin 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. doi: 10.1145/1143844.1143966.

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

Andrea Walther and Andreas Griewank. Getting started with ADOL-C. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012. doi: 10.1201/b11644-8.

Robert E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7:463–4, 1964.

Paul J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

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

David Wingate, Noah Goodman, Andreas Stuhlmüller, and Jeffrey Mark Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. *Advances in Neural Information Processing Systems*, 23, 2011.

Weiwei Yang, Yong Zhao, Li Yan, and Xiaoqian Chen. Application of PID controller based on BP neural network using automatic differentiation method. In F. Sun, J. Zhang, Y. Tan, J. Cao, and W. Yu, editors, *Advances in Neural Networks—ISNN 2008*, volume 5264 of *Lecture Notes in Computer Science*, pages 702–711. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-87734-9_80.

Ilker Yildirim, Tejas D. Kulkarni, Winrich A. Freiwald, and Joshua B. Tenenbaum. Efficient and robust analysis-by-synthesis in vision: A computational framework, behavioral tests, and modeling neuronal representations. In *Annual Conference of the Cognitive Science Society*, 2015.

Haonan Yu and Jeffrey Mark Siskind. Grounded language learning from video described with sentences. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 53–63, Sofia, Bulgaria, 2013. Association for Computational Linguistics.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In *International Conference on Machine Learning*, pages 421–429, 2016.

Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–60, 1997. doi: 10.1145/279232.279236.