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

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}	$= x_1$	$= 2$	$\boldsymbol{\bar{x}_1 = \bar{v}_{-1}}$		$\mathbf{= 5.5}$
v_0	$= x_2$	$= 5$	$\boldsymbol{\bar{x}_2 = \bar{v}_0}$		$\mathbf{= 1.716}$
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 $\text{ops}(f)$, the time it takes to calculate the $m \times n$ Jacobian by the forward mode is $n\, c\, \text{ops}(f)$, whereas the same computation can be done via reverse mode in $m\, c\, \text{ops}(f)$, 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).

Collage of reverse and forward mode of automatic differentiation

Apparently in the style of Pearlmutter and Siskind , see last page for references.

Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin(x_2)$ 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		
$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

Reverse Adjoint (Derivative) Trace		
$\bar{\boldsymbol{x}}_1 = \bar{\boldsymbol{v}}_{-1}$		$\mathbf{= 5.5}$
$\bar{\boldsymbol{x}}_2 = \bar{\boldsymbol{v}}_0$		$\mathbf{= 1.716}$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$	$= 1$	

Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin(x_2)$ 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	
$v_{-1} = x_1$	$= 2$	$\dot{v}_{-1} = \dot{x}_1$	$= 1$
$v_0 = x_2$	$= 5$	$\dot{v}_0 = \dot{x}_2$	$= 0$
$v_1 = \ln v_{-1}$	$= \ln 2$	$\dot{v}_1 = \dot{v}_{-1} / v_{-1}$	$= 1/2$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2 = \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$v_3 = \sin v_0$	$= \sin 5$	$\dot{v}_3 = \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$v_4 = v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4 = \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5 = \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$y = v_5$	$= 11.652$	$\dot{\boldsymbol{y}} = \dot{\boldsymbol{v}}_5$	$\mathbf{= 5.5}$

```
#MMLSeminar29jan19 filename=annoTraskGF.py
# rationalized version of Trask1anno.py (from 19oct18)
#last edit 31jan19

## Numpy is the Python library for array operations
import numpy as np

## To round printouts to 3 decimal places in printout

np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})

##### re-insert the options

## So outputs(y's) = fcn(inputs (x's) using  weights (w's)
## XOR operates on all 4 possible truth values, no sampling
## final 1's absorb bias
##aka linearizing affine transformations, projectivive coords, homogenizing

X0 = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])  ## X0(4,3)=4rows of 3vcrs
y0 = np.array([[0,1,1,0]]).T ## XOR, y0(4,1) column vector
np.random.seed(1) ##for repeatable experiments

W1= 2*np.random.random((3,4))-1 ##W1(3,4) signed fractions
w2= 2*np.random.random((4,1))-1 ##w2(4,1) signed fractions

print "initial W1"
print W1
print "initial w2"
print w2

for jj in xrange(600): ## by 600 obvious trend, Trask uses 60,000
    Y1 = 1/(1+np.exp(-(np.dot(X0,W1)))) #Y1(4,4) = sigma( X0(4,3)W1(3,4) )
    y2 = 1/(1+np.exp(-(np.dot(Y1,w2)))) #y3(4,1) = sigma( Y1(4,4)w2(4,1) )

## Error (aka Loss) fcn E(y2)=.5|y0-y2|^2 so -grad E = y0-y2
## dy2 for Rumelhart's deltas aka "adjoints" for updating the weights by
## backprop aka reverse automatic differentiation
    dy2= (y0-y2)*y2*(1-y2) ## dy2(4,1), Hadamard arithmetic * is termwise
    dY1= dy2.dot(w2.T)*Y1*(1-Y1) ## dY1(4,4) = dy2(4,1)w2.T(1,4)
## update the weights
    w2 += Y1.T.dot(dy2) ## dw2(4,1) = Y1.T(4,4)dy2(4,1)
    W1 += X0.T.dot(dY1) ## dW1(3,4) = X0.T(3,4)dY1(4,4)
#endfor
print "final W1="
print W1
print "final w2="
print w2
print "outcome Y1="
print Y1
print "outcome y2="
print y2
```

Notational note:
dy2 and dY1 are misleading. These socalled adjoints are not itty bitty displacements, like differentials on a computer. They are partial differentials of the error fcn 0.5 |y0-y2|^2 with respect to the variable y2 and Y1 respectively. Sorry.

Libretto for playing with the initial values of in the weights-space in R^20

```
30jan19 filename=recipeInits.txt
White shell = EDIT
Black shell = RUN, adjust !
--------------------------------
Run1: (default with random weights
1.0 adjust BLACK size, shape
1.1 run to 599
1.2 compare to 598
1.3 un-comment wait-for-keypress
1.4 by 300 obvious separation
--------------------------------
Run2: W1=w2=0  FAILS
2.1 run to 599
2.2 keypress to check for typos
NOTE the halves in output
--------------------------------
Run3: W1=0 w2=1   FAILS
3.1 keypress to check for typos
3.2 run to end
--------------------------------
Run4: W1=w2=0 contaminated
in 600 iterations FAILS
but 6000 SUCCESS
3.1 keypress to check for typos
3.2 run to end
```

filename: playInits.py

```
#28jan19 Exploring the initial conditions, corr 31,29jan19
import numpy as np ## Python library for array operations
## Round printouts to 3 decimal places
np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})

X0 = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]]) ## X0(4,3)=4rows of 3vcrs
y0 = np.array([[0,1,1,0]]).T ## XOR, y0(4,1) column vector
np.random.seed(1) ##for repeatable experiments

W1= 2*np.random.random((3,4))-1 ## W1(3,4) signed fractions
w2= 2*np.random.random((4,1))-1 ## w2(4,1) signed fractions

#eps = 0.000000000001 #.0001 ok too
#W1= 1 - eps*np.random.random((3,4)) ##W1(3,4) nhd of 1
#w2= 1 - eps*np.random.random((4,1)) ##w2(4,1) nhd of 1

#W1 = np.zeros((3,4)) ## solid 0s
#w2 = np.zeros((4,1))

#W1 = np.ones((3,4)) ## solid 1s
#w2 = np.ones((4,1))

#W1[2][3]= 1. ## contaminators
#w2[1]=-1.

for jj in xrange(600): ## by 600 obvious trend, Trask uses 60,000
    Y1 = 1/(1+np.exp(-(np.dot(X0,W1)))) #Y1(4,4)=sigma(X0(4,3)W1(3,4))
    y2 = 1/(1+np.exp(-(np.dot(Y1,w2)))) #y2(4,1)=sigma(Y1(4,4)w2(4,1))

###    raw_input() ##### wait-for-keypress kludge
    print "ITERATION", jj
    print "Weight1"
    print W1
    print "weight2"
    print w2
    print "output"
    print y2

    dy2= (y0-y2)*y2*(1-y2) ## dy2(4,1) Hadamard product *
    dY1= dy2.dot(w2.T)*Y1*(1-Y1) ## dY1(4,4) = dy2(4,1)w2.T(1,4)
    w2 += Y1.T.dot(dy2) ## w2(4,1) += Y1.T(4,4)dy2(4,1)
    W1 += X0.T.dot(dY1) ## W1(4,4) += X0.T(3,4)dY1(4,4)
#endfor
```

Play with inits by commenting in/out code lines and changing values.
Line ### waits for a [return] key to go to the next iteration. Without it
program will run through all 600 iterations.

```
http://iamtrask.github.io/2015/07/12/basic-python-network/
;;vocabulary in Trask's Python code decoded somewhat

X = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ]) ;; X(4,3) ~ 4 rows of 3 vcrs
y = np.array([[0,1,1,0]]).T ;; y(1,4).T = y.T(4,1) ~ column 4 vecr

;; synapse ~ weights,  in non-bioenvy jargon

syn0 = 2*np.random.random((3,4)) - 1 ;; syn0(3,4) of signed fractions
syn1 = 2*np.random.random((4,1)) - 1 ;; syn1(4,1) of signed fractions
for j in xrange(60000):  ;; overkill, but a matter of taste

;; the "lj" is the output of layer j

l1 = 1/(1+np.exp(-(np.dot(X,syn0)))) ;; l1(4,4)=sigma(X(4,3)syn0(3,4))
l2 = 1/(1+np.exp(-(np.dot(l1,syn1))));; l2(4,1)=sigma(l1(4,4)syn1(4,1)

;;delta ~ allusion to Rumelhart's backprop quantities
;; Python numpy library syntax uses a  mixture of
;; matrix and the commutative termwise (Hadamard) product (dot and *)
;; y-l2 ~ gradient of loss fcn = .5|y-l2|^2
;; note that for b=sigma(a), db/da = b*(1-b)

l2_delta = (y - l2)*(l2*(1-l2)) ;; backpropped to 2nd layer
l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1));; backpropped to 2nd layer
syn1 += l1.T.dot(l2_delta) ;; update the second weights
syn0 += X.T.dot(l1_delta)  ;; update the first weights

;; To make sense of this obviously "working" code I ended up spending
;; way too much time on trying to find more memorable notation,
;; but can't say I succeeded.
```

30jan19
Treppensagen aka Dits-Escaliers (there is no English equivalent)
for "Things I should have said". They occur on the way down the
stairs afterwards.

It seems that the survey paper I referred to as Baydin++ now has
a different URL than where I found it. The date is the earliest I
could find. I have added a (recent) URL prefixed with an *:

Baydin, Pearlmutter, Radul, Siskind (2017) "Automatic differentiation in machine learning"
*http://jmlr.org/papers/volume18/17-468/17-468.pdf

It is flanked by an as yet unpublished paper by Peralmutter and Siskind,
and the Wikipedia article"

Pearlmutter and Siskind "Reverse-mode AD in a functional framework"
*http://www-bcl.cs.may.ie/~barak/papers/toplas-reverse.pdf

Automatic differentiation - Wikipedia (last updated 2019)
*https://en.wikipedia.org/wiki/Automatic_differentiation

So, I think we can attribute to Pearlmutter and Siskind the convergence of
reverse mode automatic differentiation with back propagation I talked
about.