Automatic differentiation

Håvard Berland

Department of Mathematical Sciences, NTNU

September 14, 2006

Abstract

Automatic differentiation is introduced to an audience with basic mathematical prerequisites. Numerical examples show the defiency of divided difference, and dual numbers serve to introduce the algebra being one example of how to derive automatic differentiation. An example with forward mode is given first, and source transformation and operator overloading is illustrated. Then reverse mode is briefly sketched, followed by some discussion.

(45 minute talk)

What is automatic differentiation?

- Automatic differentiation (AD) is software to transform code for one function into code for the derivative of the function.

$$\begin{tikzcd}
\texttt{f(x) \{...\};} \arrow[r, "\text{Automatic differentiation}"] & \texttt{df(x) \{...\};} \\
y = f(x) \arrow[u, "\text{human programmer}"'] \arrow[r, dashed, "\text{symbolic differentiation (human/computer)}"'] & y' = f'(x) \arrow[u, dashed]
\end{tikzcd}$$

Why automatic differentiation?

Scientific code often uses both functions *and* their derivatives, for example Newtons method for solving (nonlinear) equations;

$$\text{find} \quad x \quad \text{such that} \quad f(x) = 0$$

The Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

But how to compute $f'(x_n)$ when we only know $f(x)$?

- Symbolic differentiation?
- Divided difference?
- Automatic differentiation? Yes.

Divided differences

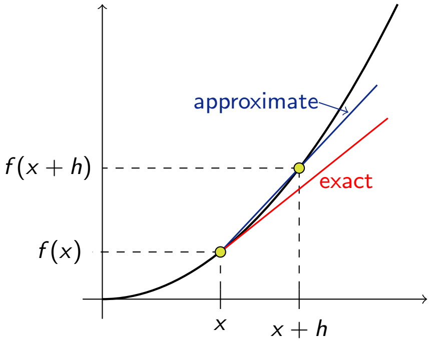
By definition, the derivative is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

so why not use

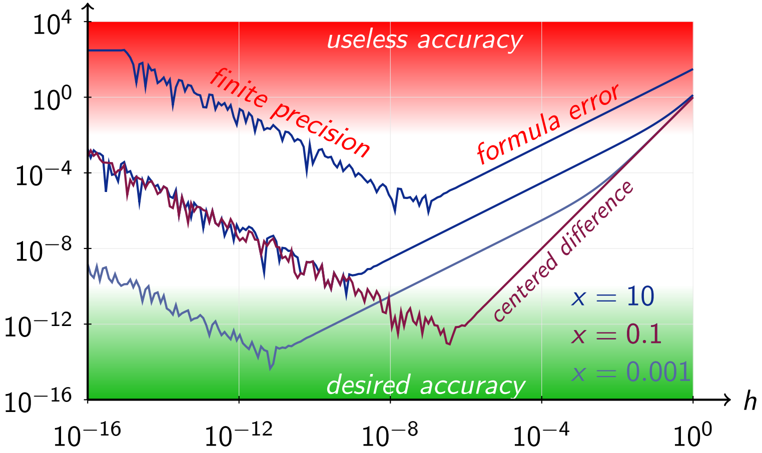
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

for some appropriately small h?



Accuracy for divided differences on $f(x) = x^3$

$$\text{error} = \left| \frac{f(x+h) - f(x)}{h} - 3x^2 \right|$$



- Automatic differentiation will ensure desired accuracy.

Dual numbers

Extend all numbers by adding a second component,

$$x \mapsto x + \dot{x}\mathbf{d}$$

- $\mathbf{d}$ is just a symbol distinguishing the second component,
- analogous to the imaginary unit $\mathbf{i} = \sqrt{-1}$.
- But, let $\mathbf{d}^2 = 0$, as opposed to $\mathbf{i}^2 = -1$.

Arithmetic on dual numbers:

$$(x + \dot{x}\mathbf{d}) + (y + \dot{y}\mathbf{d}) = x + y + (\dot{x} + \dot{y})\mathbf{d}$$

$$\begin{aligned}(x + \dot{x}\mathbf{d}) \cdot (y + \dot{y}\mathbf{d}) &= xy + x\dot{y}\mathbf{d} + \dot{x}y\mathbf{d} + \overbrace{\dot{x}\dot{y}\mathbf{d}^2}^{=0} \\ &= xy + (x\dot{y} + \dot{x}y)\mathbf{d}\end{aligned}$$

$$-(x + \dot{x}\mathbf{d}) = -x - \dot{x}\mathbf{d}, \qquad \frac{1}{x + \dot{x}\mathbf{d}} = \frac{1}{x} - \frac{\dot{x}}{x^2}\mathbf{d} \quad (x \neq 0)$$

Polynomials over dual numbers

Let

$$P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$$

and extend x to a dual number $x + \dot{x}\mathbf{d}$.
Then,

$$\begin{aligned} P(x + \dot{x}\mathbf{d}) &= p_0 + p_1(x + \dot{x}\mathbf{d}) + \cdots + p_n(x + \dot{x}\mathbf{d})^n \\ &= p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n \\ &\quad + p_1\dot{x}\mathbf{d} + 2p_2 x\dot{x}\mathbf{d} + \cdots + np_n x^{n-1}\dot{x}\mathbf{d} \\ &= P(x) + P'(x)\dot{x}\mathbf{d} \end{aligned}$$

- $\dot{x}$ may be chosen arbitrarily, so choose $\dot{x} = 1$ (currently).
- *The second component is the derivative of $P(x)$ at x*

Functions over dual numbers

Similarly, one may derive

$$
\begin{aligned}
\sin(x + \dot{x}\mathbf{d}) &= \sin(x) + \cos(x)\,\dot{x}\mathbf{d} \\
\cos(x + \dot{x}\mathbf{d}) &= \cos(x) - \sin(x)\,\dot{x}\mathbf{d} \\
e^{(x + \dot{x}\mathbf{d})} &= e^{x} + e^{x}\dot{x}\mathbf{d} \\
\log(x + \dot{x}\mathbf{d}) &= \log(x) + \frac{\dot{x}}{x}\mathbf{d} \quad x \neq 0 \\
\sqrt{x + \dot{x}\mathbf{d}} &= \sqrt{x} + \frac{\dot{x}}{2\sqrt{x}}\mathbf{d} \quad x \neq 0
\end{aligned}
$$

Conclusion from dual numbers

Derived from dual numbers:

A function applied on a dual number will return its *derivative* in the second/dual component.

We can extend to functions of many variables by introducing more dual components:

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

extends to

$$\begin{aligned} f(x_1 + \dot{x}_1\mathbf{d}_1, x_2 + \dot{x}_2\mathbf{d}_2) &= \\ &(x_1 + \dot{x}_1\mathbf{d}_1)(x_2 + \dot{x}_2\mathbf{d}_2) + \sin(x_1 + \dot{x}_1\mathbf{d}_1) = \\ &\qquad x_1x_2 + (x_2 + \cos(x_1))\dot{x}_1\mathbf{d}_1 + x_1\dot{x}_2\mathbf{d}_2 \end{aligned}$$

where $\mathbf{d}_i\mathbf{d}_j = 0$.

Decomposition of functions, the chain rule

Computer code for $f(x_1, x_2) = x_1 x_2 + \sin(x_1)$ might read

Original program

$$\begin{aligned} w_1 &= x_1 \\ w_2 &= x_2 \\ w_3 &= w_1 w_2 \\ w_4 &= \sin(w_1) \\ w_5 &= w_3 + w_4 \end{aligned}$$

Dual program

$$\begin{aligned} \dot{w}_1 &= 0 \\ \dot{w}_2 &= 1 \\ \dot{w}_3 &= \dot{w}_1 w_2 + w_1 \dot{w}_2 = 0 \cdot x_2 + x_1 \cdot 1 = x_1 \\ \dot{w}_4 &= \cos(w_1)\dot{w}_1 = \cos(x_1) \cdot 0 = 0 \\ \dot{w}_5 &= \dot{w}_3 + \dot{w}_4 \quad = x_1 + 0 = x_1 \end{aligned}$$

and

$$\frac{\partial f}{\partial x_2} = x_1$$

The chain rule

$$\frac{\partial f}{\partial x_2} = \frac{\partial f}{\partial w_5} \frac{\partial w_5}{\partial w_3} \frac{\partial w_3}{\partial w_2} \frac{\partial w_2}{\partial x_2}$$

ensures that we can *propagate* the dual components throughout the computation.

Realization of automatic differentiation

Our current procedure:

1. Decompose original code into intrinsic functions
2. Differentiate the intrinsic functions, effectively symbolically
3. Multiply together according to the chain rule

How to “automatically” transform the “original program” into the “dual program”?

Two approaches,

- Source code transformation (C, Fortran 77)
- Operator overloading (C++, Fortran 90)

Source code transformation by example

function.c

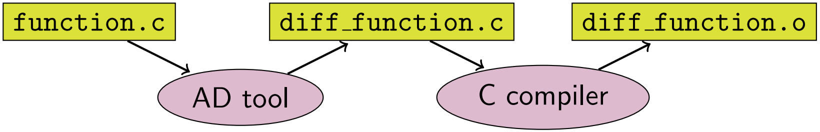
```
double  f(double x1, double x2) {
  double w3, w4, w5;
  w3 = x1 * x2;

  w4 = sin(x1);

  w5 = w3 + w4;



  return w5;
}
```

function.c

Source code transformation by example

diff_function.c

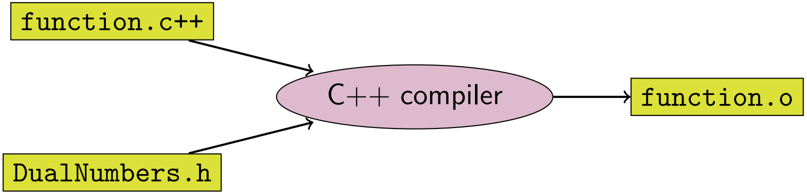
```
double* f(double x1, double x2, double dx1, double dx2) {
  double w3, w4, w5, dw3, dw4, dw5, df[2];
  w3 = x1 * x2;
  dw3 = dx1 * x2 + x1 * dx2;
  w4 = sin(x1);
  dw4 = cos(x1) * dx1;
  w5 = w3 + w4;
  dw5 = dw3 + dw4;
  df[0] = w5;
  df[1] = dw5;
  return df;
}
```



Operator overloading

function.c++

```
Number f(Number x1, Number x2) {
  w3 = x1 * x2;
  w4 = sin(x1);
  w5 = w3 + w4;
  return w5;
}
```



Source transformation vs. operator overloading

Source code transformation:

+ Possible in all computer languages
+ Can be applied to your old legacy Fortran/C code.
+ Allows easier compile time optimizations.

÷ Source code swell

÷ More difficult to code the AD tool

Operator overloading:

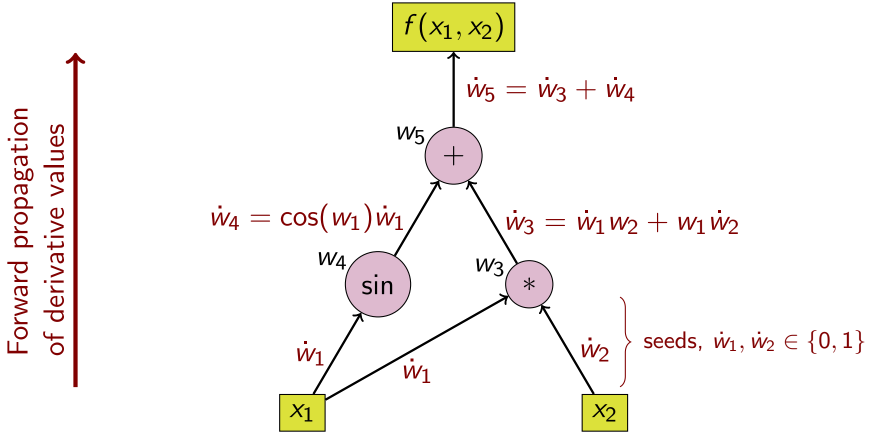
+ No changes in your original code
+ Flexible when you change your code or tool
+ Easy to code the AD tool

÷ Only possible in selected languages

÷ Current compilers lag behind, code runs slower

Forward mode AD

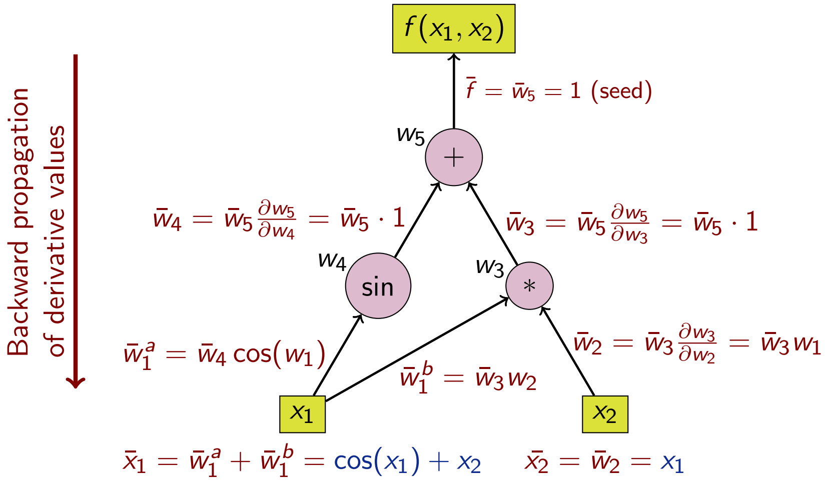
- We have until now only described *forward mode* AD.
- Repetition of the procedure using the computational graph:



Forward propagation of derivative values
$f(x_1, x_2)$
$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$
w_5
$+$
$\dot{w}_4 = \cos(w_1)\dot{w}_1$
$\dot{w}_3 = \dot{w}_1 w_2 + w_1 \dot{w}_2$
w_4
sin
w_3
$*$
$\dot{w}_1$
$\dot{w}_1$
$\dot{w}_2$
seeds, $\dot{w}_1, \dot{w}_2 \in \{0, 1\}$
x_1
x_2

Reverse mode AD

- The chain rule works in both directions.
- The computational graph is now traversed from the top.



$f(x_1, x_2)$
$\bar{f} = \bar{w}_5 = 1$ (seed)
w_5
$+$
$\bar{w}_4 = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = \bar{w}_5 \cdot 1$
$\bar{w}_3 = \bar{w}_5 \frac{\partial w_5}{\partial w_3} = \bar{w}_5 \cdot 1$
w_4
sin
w_3
$*$
Backward propagation of derivative values
$\bar{w}_1^a = \bar{w}_4 \cos(w_1)$
$\bar{w}_1^b = \bar{w}_3 w_2$
$\bar{w}_2 = \bar{w}_3 \frac{\partial w_3}{\partial w_2} = \bar{w}_3 w_1$
x_1
x_2
$\bar{x}_1 = \bar{w}_1^a + \bar{w}_1^b = \cos(x_1) + x_2$
$\bar{x}_2 = \bar{w}_2 = x_1$

Jacobian computation

Given $F\colon \mathbf{R}^n \mapsto \mathbf{R}^m$ and the Jacobian $J = DF(\mathbf{x}) \in \mathbf{R}^{m\times n}$.

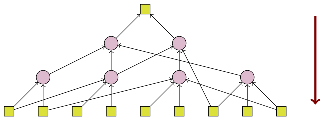
$$J = DF(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- One sweep of forward mode can calculate one column vector of the Jacobian, $J\dot{\mathbf{x}}$, where $\dot{\mathbf{x}}$ is a column vector of seeds.
- One sweep of reverse mode can calculate one row vector of the Jacobian, $\bar{\mathbf{y}}J$, where $\bar{\mathbf{y}}$ is a row vector of seeds.
- Computational cost of one sweep forward or reverse is roughly equivalent, but reverse mode requires access to *intermediate* variables, requiring more memory.

Forward or reverse mode AD?

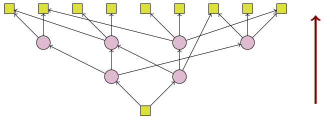
Reverse mode AD is best suited for

$$F\colon \mathbf{R}^n \to \mathbf{R}$$

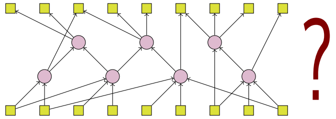


Forward mode AD is best suited for

$$G\colon \mathbf{R} \to \mathbf{R}^m$$



- Forward and reverse mode represents just two possible (extreme) ways of recursing through the chain rule.
- For $n > 1$ and $m > 1$ there is a golden mean, but finding the optimal way is probably an *NP*-hard problem.



Discussion

- Accuracy is guaranteed and complexity is not worse than that of the original function.
- AD works on iterative solvers, on functions consisting of thousands of lines of code.
- AD is trivially generalized to higher derivatives. Hessians are used in some optimization algorithms. Complexity is quadratic in highest derivative degree.
- The alternative to AD is usually symbolic differentiation, or rather using algorithms not relying on derivatives.
- Divided differences may be just as good as AD in cases where the underlying function is based on discrete or measured quantities, or being the result of stochastic simulations.

Applications of AD

- Newton's method for solving nonlinear equations
- Optimization (utilizing gradients/Hessians)
- Inverse problems/data assimilation
- Neural networks
- Solving stiff ODEs

For software and publication lists, visit

- `www.autodiff.org`

Recommended literature:

- Andreas Griewank: *Evaluating Derivatives*. SIAM 2000.