Automatic differentiation

Note Title 2025-03-28

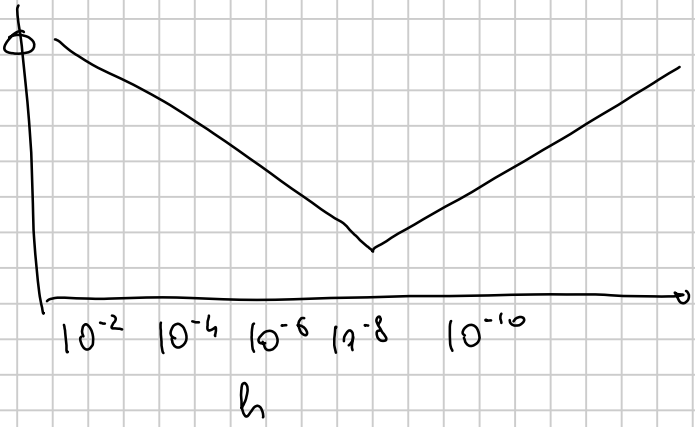
We have code to compute $y = f(x)$ $\quad f: \mathbb{R}^m \to \mathbb{R}^n$

Can we also compute $f'(x)$ without having to write code for it (and obtain formulas).

First classical approach: numerical differentiation $\quad f: \mathbb{R} \to \mathbb{R}$

- set $h > 0$
- compute $g = \dfrac{f(x+h) - f(x)}{h}$

Accuracy



Analytical error:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(\xi)h^2$$

$$\frac{f(x+h) - f(x)}{h} - f'(x) = \frac{1}{2}f''(\xi)h$$

$$|g - f'(x)| = \frac{1}{2}f''(\xi)h$$

Machine error:

$f(x)(1+\varepsilon_1)$ $\quad |\varepsilon_1| \le u$ even with the best

$f(x+h)(1+\varepsilon_2)$ $\quad |\varepsilon_2| \le u$ algorithm in the world

$$\tilde{g} = \frac{f(x+h)(1+\varepsilon_1) - f(x)(1+\varepsilon_2)}{h}$$

$$|\tilde{g} - g| \le \left|\frac{1}{h} f(x+h)\right| u + \left|\frac{1}{h} f(x)\right| u$$

$$|\tilde{g} - f'(x)| \le |\tilde{g} - g| + |g - f'(x)| \le \frac{1}{2} f''(\xi) h + \frac{|f(x+h)| + |f(x)|}{h} u$$

minimum error when $\frac{1}{2} f''(\xi) h = \frac{|f(x+h)| + |f(x)|}{h} u$

$h \approx u^{1/2}$ gives error $u^{1/2}$

Ex: do the same analysis for

$$g = \frac{f(x+h) - f(x-h)}{2h}$$

error $\approx u^{2/3} \sim 10^{-10} - 10^{-11}$

Idea: we can get better, if we write our function to support also matrix arguments.

square

then, $f(J_\lambda) = \begin{bmatrix} f(\lambda) & f'(\lambda) & \cdots & \frac{1}{(k-1)!} f^{(k-1)}(\lambda) \\ & \ddots & \ddots & \vdots \\ & & \ddots & f'(\lambda) \\ 0 & & & f(\lambda) \end{bmatrix}$

Automatic differentiation

- full machine precision
- works for a wide variety of constructs

Change code so that it works on matrix arguments:

```
z = x
while |z| < 5
    z = z + 2
end
```

⇒

```
z = x
while |z(1,1)| < 5
    z = z + 2*eye(n)
end
```

$y = e^z$ $\quad$ $y = \text{expm}(z)$

If we want to compute the function on a Jordan block of size $n=3$, all the matrices we encounter are of the form

$$\begin{bmatrix} a & b & c \\ 0 & a & b \\ 0 & 0 & a \end{bmatrix} = a \cdot I + b \cdot E + c \cdot E^2 \qquad E = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

We can treat E as an indeterminate, and compute

$f(5+E)$ in the polynomial field $\dfrac{\mathbb{R}[E]}{(E^3)}$

This lowers the cost from $O(n^3)$
to $O(n^2)$ (naive polynomial arithmetic)
$O(n \log n)$ $(F \neq T)$

From an analysis point of view:
start computing from $x = 5+\varepsilon$ ← $[5\ 1\ 0]$

$w = x^2 = (5+\varepsilon)(5+\varepsilon) = 25 + 10\varepsilon + \varepsilon^2 + O(\varepsilon^3)$ ← $[5\ 1\ 0] * [5\ 1\ 0] = [25\ 10\ 1]$

$z = x + 5 = 10 + \varepsilon + O(\varepsilon^3)$

$y = wz = (25 + 10\varepsilon + \varepsilon^2)(10+\varepsilon) = 250 + 125\varepsilon + 20\varepsilon^2 + O(\varepsilon^3)$

The result gives the power series expansion of y in terms of ε, and this reveals derivatives.

We can implement this in code using object-oriented programming.

Idea: define new types of variables (classes)
that contain data (members)
and functions/operations that operate on them (methods)

A Taylor expansion of degree 2 can be represented by a vector: $5+\varepsilon \to [5\ 1\ 0]$

Operations to define:

- $[a_0\ a_1\ a_2] * [b_0\ b_1\ b_2] = [a_0b_0\ \ a_1b_0 + a_0b_1\ \ a_2b_0 + a_1b_1 + a_0b_2]$
- $[a_0\ a_1\ a_2] + [b_0\ b_1\ b_2] = [a_0+b_0\ \ a_1+b_1\ \ a_2+b_2]$
- $[a_0\ a_1\ a_2] + k = [a_0\ a_1\ a_2] + [k\ 0\ 0]$ for $k \in \mathbb{R}$.

Once these are defined, $f(\text{Taylor}([5\ 1\ 0]))$ returns f with its derivatives.

exp(Taylor) can be defined via

$$\exp(a+b\varepsilon+c\varepsilon^2) = \sum_{k\geq 0} \frac{1}{k!}(a+b\varepsilon+c\varepsilon^2)^k$$

If there is only one derivative:

a with derivative a'

$b = \exp(a) \qquad b' = \exp(a)\,a'$

Generally:

$z = f(a, b, \ldots)$

$z' = \frac{\partial f}{\partial a}a' + \frac{\partial f}{\partial b}b' + \cdots$

$z'' = \frac{\partial^2 f}{\partial a^2}(a')^2 + \frac{\partial f}{\partial a}a'' + \frac{\partial^2 f}{\partial b^2}(b')^2 + \frac{\partial f}{\partial b}b'' + \cdots$

more complicated expressions for further derivatives.

The $n=2$ (only one derivative) case is known as the dual numbers:

The dual numbers are the ring $\frac{\mathbb{R}[\varepsilon]}{(\varepsilon^2)}$

e.g. $(5+\varepsilon)(2+3\varepsilon) = 10 + 17\varepsilon$

With multivariate functions, this still works:

$z = f(a, b, \dots)$

$\downarrow$

$z' = \frac{\partial f}{\partial a} a' + \frac{\partial f}{\partial b} b'$

Jacobians

matrix multiplication

$\frac{dx}{dx} = I$, then for each intermediate variable z compute $\frac{dz}{dx} \in \mathbb{R}^{n_z \times n}$

$c = a \cdot b \qquad \frac{dc}{dx} = a \frac{db}{dx} + \frac{da}{dx} \cdot b$

The cost depends on the dimensions of the involved matrix multiplications

If $x \in \mathbb{R}^n$, n intermediate variables are of dimension n, $y \in \mathbb{R}^m$, $m < n$, the cost is $O(n^3)$

This method is more effective when $f: \mathbb{R}^n \to \mathbb{R}^m$, with $n \leq m$, and all intermediate variables of dimension n.

Neural networks: a big fitting problem $\min L(w)$

where $w \in \mathbb{R}^n$, n very large, $L: \mathbb{R}^n \to \mathbb{R}$.

Since n (and intermediate variables) are very large, this is inefficient!

Idea to circumvent the problem: start from the last variable: "reverse mode" of automatic differentiation.

Take our model problem

$w = x^2$ input: x

$z = x + 5$ output: y

$y = w \cdot z$

In the usual "forward mode", we compute derivatives w.r.t. x for each variable:

$\frac{dx}{dx} = 1$ $\frac{dw}{dx}$, $\frac{dz}{dx}$, $\frac{dy}{dx}$

In reverse mode, we try to start from the end:

$\frac{dy}{dy} = 1$, $\frac{dy}{dw}$, $\frac{dy}{dz}$, $\frac{dy}{dx}$

(In a multivariate context where $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $n >> m$, these are "small" $m \times$ something matrices)