Introduction to Automatic Differentiation

Bogdan Burlacu

University of Applied Sciences Upper Austria
Heuristic and Evolutionary Algorithms Laboratory

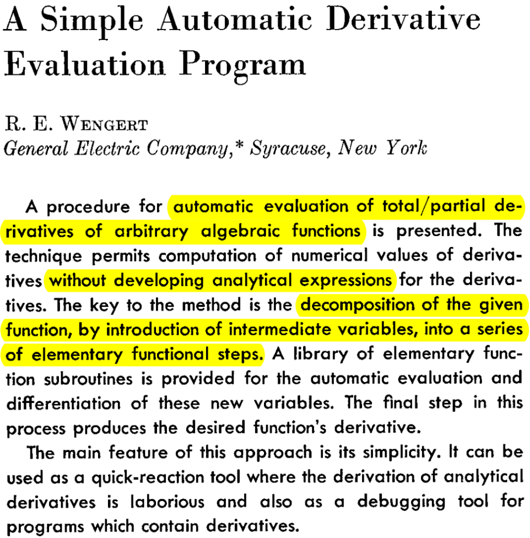
Slides and demo code

https://github.com/foolnotion/reverse-ad-demo

Autodiff Workshop, NIPS 2016

`https://autodiff-workshop.github.io/2016.html`

1. Introduction
2. The Chain Rule
3. Forward and Reverse Mode
4. Computational Graph Examples
5. Implementation
6. Benchmarks
7. Conclusion



A Simple Automatic Derivative Evaluation Program

R. E. Wengert
General Electric Company, Syracuse, New York*

A procedure for automatic evaluation of total/partial derivatives of arbitrary algebraic functions is presented. The technique permits computation of numerical values of derivatives without developing analytical expressions for the derivatives. The key to the method is the decomposition of the given function, by introduction of intermediate variables, into a series of elementary functional steps. A library of elementary function subroutines is provided for the automatic evaluation and differentiation of these new variables. The final step in this process produces the desired function's derivative.

The main feature of this approach is its simplicity. It can be used as a quick-reaction tool where the derivation of analytical derivatives is laborious and also as a debugging tool for programs which contain derivatives. [1]

[1] [Wengert, 1964]

Arguably, the subject began with Newton and Leibniz[2]

G.W. Leibniz

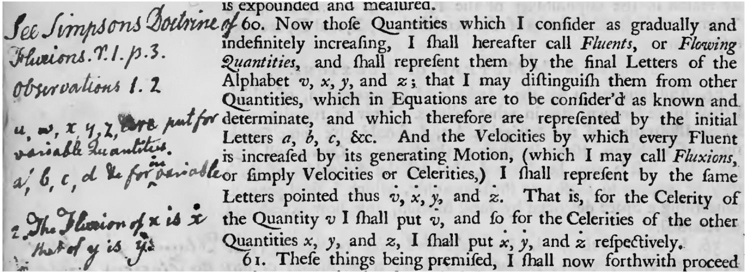
- Nova methodvs pro maximis et minimis, itemque tangentibus, quae nec fractas, nec irrationales quantitates moratur, et singulare pro illis calculi genus.
  A new method for maxima and minima as well as tangents, which is impeded neither by fractional nor irrational quantities, and a remarkable type of calculus for them. (1684)
- Machina arithmetica in qua non additio tantum et subtractio sedet multiplicatio nullo, division veropaene nullo animi labore peragantur.
  An arithmetic machine which can be used to carry out not only addition and subtraction but also multiplication with no, and division with really almost no, intellectual exertion. (1685)

I. Newton

- The method of fluxions and infinite series with its application to the geometry of curve-lines
  `https://archive.org/details/methodfluxionsa00newtgoog/mode/2up`

[2] With some controversy [Christianson, 2012]

Arguably, the subject began with Newton and Leibniz[2]



See Simpsons Doctrine of
Fluxions. V.1. p.3.
Observations 1. 2

u, w, x, y, z, are put for
variable Quantities.
a, b, c, d &c for invariable

2. The Fluxion of x is $\dot{x}$
that of y is $\dot{y}$.

is expounded and meaſured.

60. Now thoſe Quantities which I conſider as gradually and indefinitely increaſing, I ſhall hereafter call *Fluents*, or *Flowing Quantities*, and ſhall repreſent them by the final Letters of the Alphabet v, x, y, and z; that I may diſtinguiſh them from other Quantities, which in Equations are to be conſider'd as known and determinate, and which therefore are repreſented by the initial Letters a, b, c, &c. And the Velocities by which every Fluent is increaſed by its generating Motion, (which I may call *Fluxions*, or ſimply Velocities or Celerities,) I ſhall repreſent by the ſame Letters pointed thus $\dot{v}$, $\dot{x}$, $\dot{y}$, and $\dot{z}$. That is, for the Celerity of the Quantity v I ſhall put $\dot{v}$, and ſo for the Celerities of the other Quantities x, y, and z, I ſhall put $\dot{x}$, $\dot{y}$, and $\dot{z}$ reſpectively.

61. Theſe things being premiſed, I ſhall now forthwith proceed

(I. Newton)

[2] With some controversy [Christianson, 2012]

Arguably, the subject began with Newton and Leibniz[2]

> *Knowing thus the **Algorithm** (as I may say) of this calculus, which I call **differential** calculus, all other differential equations can be solved by a common method. [...] For any other quantity (not itself a term, but contributing to the formation of the term) we use its differential quantity to form the differential quantity of the term itself, not by simple substitution, but according to the prescribed Algorithm (G.W. Leibniz)*

[2] With some controversy [Christianson, 2012]

...with contributions from many others

> *Beda (1959) – Wengert (1964) – Wanner (1969) – Ostrowski (1971) – Bennet (1973) – Warner (1975) – Linnainmaa (1976) – Kedem (1980) – Speelpenning (1980) – Rall (1981) – Baur and Strassen (1984) – Griewank (1989) – Bischof and Carle (1991)*

In particular, Andreas Griewank considered a central figure in the “modern rebirth” of AD (mainly due to work on reverse mode).

During this time, backpropagation was invented and reinvented within the ML community [Parker, 1985, Rumelhart et al., 1986, Werbos, 1974]

Theory from both fields was unified later [Hecht-Nielsen, 1989]

Stigler’s Law of Eponymy

No scientific law is named after its original discoverer.

Cheap gradient principle

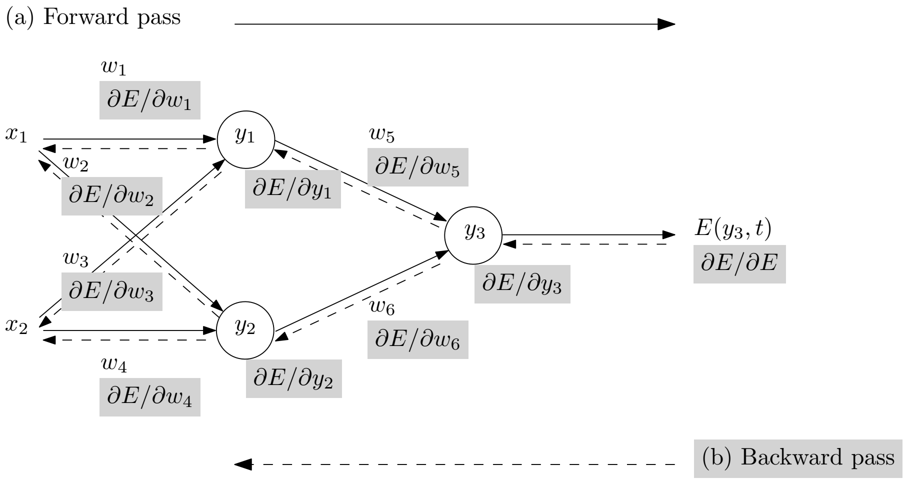
> *There is a common misconception that calculating a function of n variables and its gradient is about* $(n+1)$ *times as expensive as just calculating the function [...]*
>
> *If care is taken in handling quantities, which are common to the function and its derivatives, the ratio is usually* 1.5, *not* $(n+1)$. *(Phil Wolfe, 1982)*

$$\frac{\text{Cost(Optimization)}}{\text{Cost(Simulation)}} \sim \mathcal{O}(1)$$

Control theory saw the first application of gradient descent to large-scale optimization

There is no cheap Jacobian principle

Gradient-based optimization is a pillar of machine learning


(a) Forward pass
w_1
$\partial E/\partial w_1$
x_1
y_1
w_2
$\partial E/\partial w_2$
w_5
$\partial E/\partial w_5$
$\partial E/\partial y_1$
y_3
$E(y_3, t)$
$\partial E/\partial E$
w_3
$\partial E/\partial w_3$
$\partial E/\partial y_3$
w_6
x_2
y_2
$\partial E/\partial w_6$
w_4
$\partial E/\partial w_4$
$\partial E/\partial y_2$
(b) Backward pass

Backpropagation – image source [Baydin et al., 2017]

Automatic differentiation application domains

- Atmospheric chemistry
- Computational finance
- Computational fluid dynamics
- Differentiable ray tracing
- Engineering design and optimization
- Machine learning
- Model uncertainty and sensitivity

Deep learning ⟶ Differentiable programming

- ANNs are assembled from building blocks and trained with backpropagation
- Make algorithmic elements continuous and differentiable
- Even with complex architectures, differentiability means they can be trained end-to-end [Wang et al., 2019]
- AD support baked in languages and compilers

Function

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$$

Jacobian

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}_{m \times n}$$

J_f captures the rate of change of each component of f with respect to each component of input variable $x \in \mathbb{R}^n$

Methods

1. *Analytical*
    - Work out derivatives by hand
    - Time-consuming and error prone

Methods

1. *Analytical*
   - Work out derivatives by hand
   - Time-consuming and error prone
2. *Numerical*
   - Uses the limit definition of the derivative
   - Simple but inexact, prone to approximation errors
   - Complexity linear in the number of variables

Methods

1. *Analytical*
    - Work out derivatives by hand
    - Time-consuming and error prone
2. *Numerical*
    - Uses the limit definition of the derivative
    - Simple but inexact, prone to approximation errors
    - Complexity linear in the number of variables
3. *Symbolic*
    - Symbolic manipulation to derive closed-form expressions
    - Complex and cryptic expressions, prone to “expression swell”

Methods

1. *Analytical*
    - Work out derivatives by hand
    - Time-consuming and error prone
2. *Numerical*
    - Uses the limit definition of the derivative
    - Simple but inexact, prone to approximation errors
    - Complexity linear in the number of variables
3. *Symbolic*
    - Symbolic manipulation to derive closed-form expressions
    - Complex and cryptic expressions, prone to “expression swell”
4. *Automatic (algorithmic!)*
    - Function is decomposed into sequence of elementary operations
    - Relies on the application of the chain rule of calculus
    - Full expressive capability of the host language (loops, conditionals, ...)
    - Prevents expression swell, *exact* (up to machine precision)

Chain rule

If f is a composite function

$$\begin{aligned} g &: \mathbb{R}^k \longrightarrow \mathbb{R}^m \\ h &: \mathbb{R}^n \longrightarrow \mathbb{R}^k \\ f &: \mathbb{R}^n \longrightarrow \mathbb{R}^m \\ f &= g \circ h = g\left(h(x)\right) \end{aligned}$$

then, according to the chain rule:

$$J_{ij} = \frac{\partial f_i}{\partial x_j} = \frac{\partial g_i}{\partial h_1}\frac{\partial h_1}{\partial x_j} + \frac{\partial g_i}{\partial h_2}\frac{\partial h_2}{\partial x_j} + \ldots + \frac{\partial g_i}{\partial h_k}\frac{\partial h_k}{\partial x_j}$$

“if a car travels twice as fast as a bicycle and the bicycle is four times as fast as a walking man, then the car travels 2 × 4 = 8 times as fast as the man.”

Forward mode

- Traverse the chain rule from input to output (more "natural")
- Evaluation order coincides with flow of derivative information

$$\dot{v}_k = \frac{\partial v_k}{\partial x_i}, \; i = 1, \ldots, n$$

- Values computed in "lockstep" with no additional memory
- Each forward pass

$$\dot{x}_i = 1$$
$$\dot{x}_j = 0, \forall j \neq i$$

- Usually implemented with *dual numbers* [Clifford, 1871]
  http://ceres-solver.org/automatic_derivatives.html#dual-numbers-jets

Reverse mode

- Traverse the chain rule from output to input
- Requires "tape" to store intermediate values
- Derivative of dependent variable w.r.t. intermediate variable

$$\bar{v}_k = \frac{\partial f_j}{\partial v_k}, j = 1, \ldots, m$$

- Two-pass process
  1. forward pass: populate intermediate values
  2. reverse pass: propagate adjoints
- According to the chain rule

$$\bar{v}_k = \sum \bar{v}_l \frac{\partial v_l}{\partial v_k}, \forall l, v_k \prec v_l$$

[Griewank and Walther, 2008] (chapter 4, sections 4.5 and 4.6):

$$\text{ops}(\textit{forward}) \approx c_1 \cdot \text{ops}(f), c_1 \in [2, 2.5]$$
$$\text{ops}(\textit{reverse}) \approx c_2 \cdot \text{ops}(f), c_2 \in [3, 4]$$

The time it takes to calculate a $m \times n$ Jacobian:

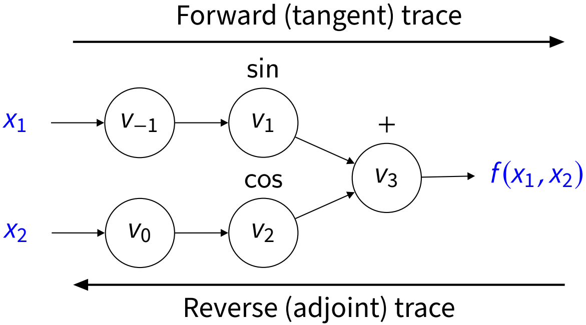
- $n \cdot c \cdot \text{ops}(f)$ in forward mode
- $m \cdot c \cdot \text{ops}(f)$ in reverse mode

Machine learning

- ∇ of a scalar-valued objective w.r.t. a large # of parameters is typically required
- this establishes reverse mode as the principal approach

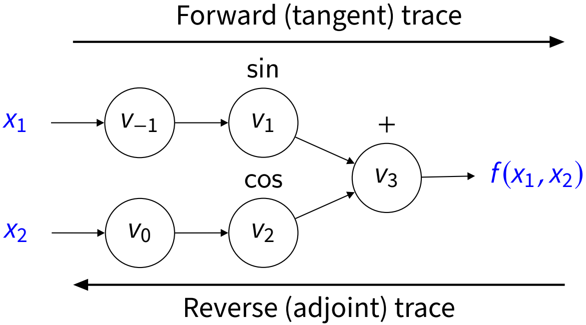
Computational graph [Bauer, 1974]

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



Computational graph [Bauer, 1974]

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

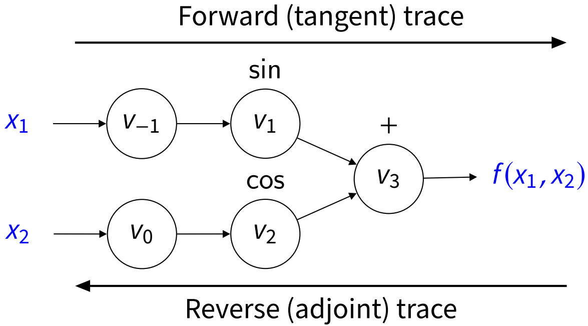


Notation [Walther, 2009]

- 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
- variables $y_{m-i} = v_{l-i}, i = m-1, \ldots, 0$ are the output variables

Computational graph [Bauer, 1974]

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



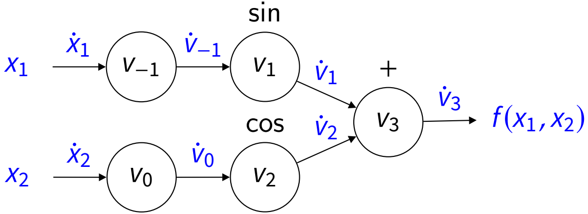
Forward (tangent) trace

Reverse (adjoint) trace

Primal trace at $(x_1, x_2) = (2, 3)$

$$\begin{array}{lclclcl}
v_{-1} & = & x_1 & = & 2 & & \\
v_0 & = & x_2 & = & 3 & & \\
v_1 & = & \sin(v_{-1}) & = & \sin(2) & = & 0.909 \\
v_2 & = & \cos(v_0) & = & \cos(3) & = & -0.990 \\
v_3 & = & v_1 + v_2 & = & 0.909 - 0.990 & = & -0.081 \blacksquare
\end{array}$$

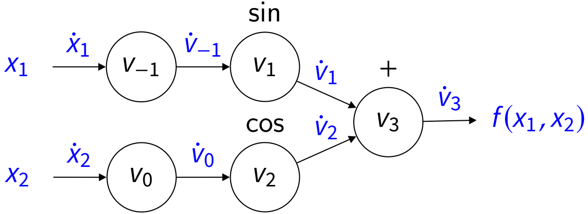
$f(x_1, x_2) = \sin(x_1) + \cos(x_2)$



Forward trace at $(x_1, x_2) = (2, 3)$ for $\dot{x}_1$

$$\begin{aligned}
\dot{v}_{-1} &= \dot{x}_1 &&= 1 \\
\dot{v}_0 &= \dot{x}_2 &&= 0 \\
\dot{v}_1 &= \dot{v}_{-1} \cdot \cos(v_{-1}) &&= 1 \cdot \cos(2) &&= -0.416 \\
\dot{v}_2 &= \dot{v}_0 \cdot - \sin(v_0) &&= 0 \cdot - \sin(3) &&= 0 \\
\dot{v}_3 &= \dot{v}_1 + \dot{v}_2 &&= -0.416 + 0 &&= -0.416 \blacksquare
\end{aligned}$$

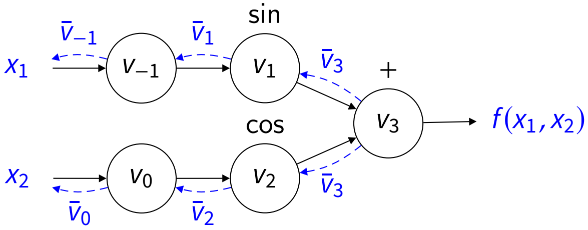
$f(x_1, x_2) = \sin(x_1) + \cos(x_2)$



Forward trace at $(x_1, x_2) = (2, 3)$ for $\dot{x}_2$

$$\begin{aligned}
\dot{v}_0 &= \dot{x}_1 &&= 0 \\
\dot{v}_1 &= \dot{x}_2 &&= 1 \\
\dot{v}_2 &= \dot{v}_0 \cdot \cos(v_0) &&= 0 \cdot \cos(2) &&= 0 \\
\dot{v}_3 &= \dot{v}_1 \cdot -\sin(v_1) &&= 1 \cdot -\sin(3) &&= -0.141 \\
\dot{v}_4 &= \dot{v}_2 + \dot{v}_3 &&= 0 - 0.141 &&= -0.141 \;\blacksquare
\end{aligned}$$

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

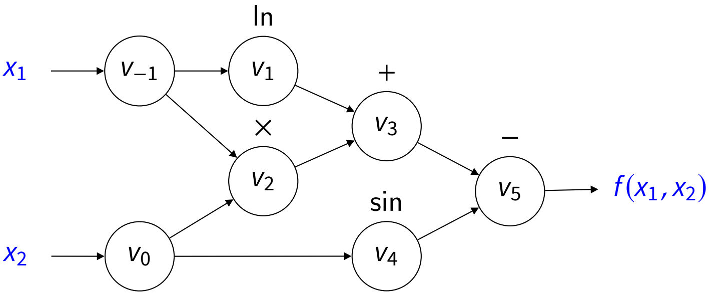


Reverse trace at $(x_1, x_2) = (2, 3)$

$$\begin{array}{lllllll}
\bar{v}_3 & = & 1 & & & & \\
\bar{v}_2 & = & \bar{v}_3\ \partial v_3/\partial v_2 & = & 1 \cdot 1 & = & 1 \\
\bar{v}_1 & = & \bar{v}_3\ \partial v_3/\partial v_1 & = & 1 \cdot 1 & = & 1 \\
\bar{v}_0 & = & \bar{v}_2\ \partial v_2/\partial v_0 & = & 1 \cdot -\sin(3) & = & -0.141 \blacksquare \\
\bar{v}_{-1} & = & \bar{v}_1\ \partial v_1/\partial v_{-1} & = & 1 \cdot \cos(2) & = & -0.416 \blacksquare
\end{array}$$

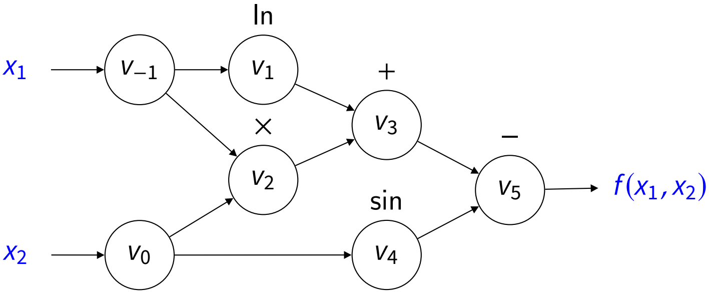
Computational graph

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



[Baydin et al., 2017]

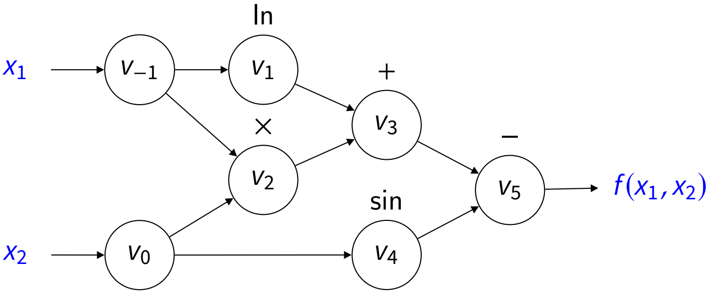
$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$



Primal trace at $(x_1, x_2) = (2, 5)$

$$\begin{aligned}
v_{-1} &= x_1 &&= 2 \\
v_0 &= x_2 &&= 5 \\
v_1 &= \ln(v_{-1}) &&= \ln(2) &&= 0.693 \\
v_2 &= v_{-1} v_0 &&= 2 \cdot 5 &&= 10 \\
v_3 &= v_1 + v_2 &&= 0.693 + 10 &&= 10.693 \\
v_4 &= \sin(v_0) &&= \sin(5) &&= -0.959 \\
v_5 &= v_3 - v_4 &&= 10.693 + 0.959 &&= 11.652 \blacksquare
\end{aligned}$$

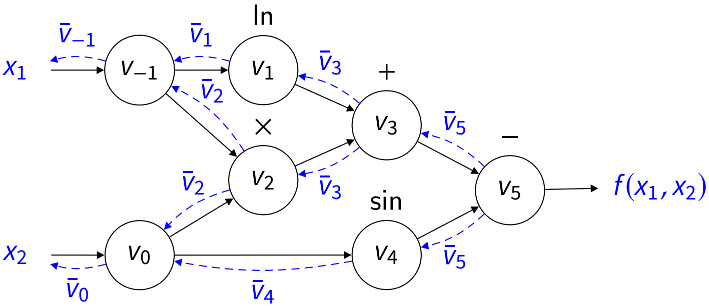
$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



Forward trace at $(x_1, x_2) = (2, 5)$ for $\dot{x}_1$

$$
\begin{array}{llllll}
\dot{v}_{-1} & = & \dot{x}_1 & = & 1 & \\
\dot{v}_0 & = & \dot{x}_2 & = & 0 & \\
\dot{v}_1 & = & \dot{v}_{-1}/v_{-1} & = & 0.5 & \\
\dot{v}_2 & = & \dot{v}_{-1}v_0 + \dot{v}_1 v_{-1} & = & 1 \cdot 5 + 0 \cdot 2 & = \quad 5 \\
\dot{v}_3 & = & \dot{v}_1 + \dot{v}_2 & = & 0.5 + 5 & = \quad 5.5 \\
\dot{v}_4 & = & \dot{v}_0 \cdot \cos(v_4) & = & 0 \cdot \cos(5) & = \quad 0 \\
\dot{v}_5 & = & \dot{v}_3 - \dot{v}_4 & = & 5.5 - 0 & = \quad 5.5 \blacksquare
\end{array}
$$

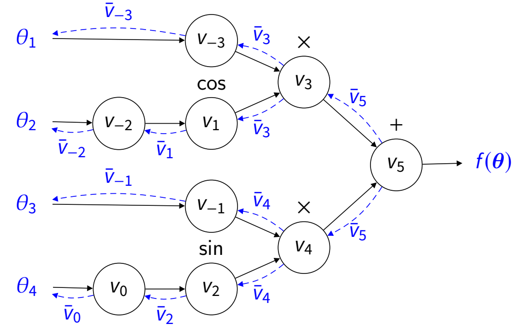
$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



Reverse trace at $(x_1, x_2) = (2, 5)$

$$
\begin{aligned}
\bar{v}_5 &= \bar{f} && = 1 \\
\bar{v}_4 &= \bar{v}_5\, \partial v_5/\partial v_4 && = 1 \cdot (-1) && = -1 \\
\bar{v}_3 &= \bar{v}_5\, \partial v_5/\partial v_3 && = 1 \cdot 1 && = 1 \\
\bar{v}_2 &= \bar{v}_3\, \partial v_3/\partial v_2 && = 1 \cdot 1 && = 1 \\
\bar{v}_1 &= \bar{v}_3\, \partial v_3/\partial v_1 && = 1 \cdot 1 && = 1 \\
\bar{v}_0 &= \bar{v}_2\, \partial v_2/\partial v_0 + \bar{v}_4\, \partial v_4/\partial v_0 && = v_{-1} - \cos(v_1) && = 1.716 \blacksquare \\
\bar{v}_{-1} &= \bar{v}_2\, \partial v_2/\partial v_{-1} + \bar{v}_1\, \partial v_1/\partial v_{-1} && = v_0 + 1/v_{-1} && = 5.5 \blacksquare
\end{aligned}
$$

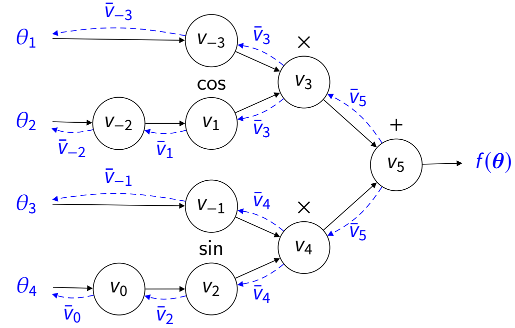
$$f(\boldsymbol{\theta}) = \theta_1 \cos(\theta_2) + \theta_3 \sin(\theta_4)$$



Primal trace at $\boldsymbol{\theta} = \begin{bmatrix} 0.5 & 2 & 0.7 & 3 \end{bmatrix}$

$$\begin{aligned}
v_{-3} &= \theta_1 &&= 0.5 \\
v_{-2} &= \theta_2 &&= 2 \\
v_{-1} &= \theta_3 &&= 0.7 \\
v_0 &= \theta_4 &&= 3 \\
v_1 &= \cos(v_{-2}) &&= -0.416 \\
v_2 &= \sin(v_0) &&= 0.141 \\
v_3 &= v_{-3} \cdot v_1 &&= -0.208 \\
v_4 &= v_{-1} \cdot v_2 &&= 0.099 \\
v_5 &= v_3 + v_4 &&= -0.109 \blacksquare
\end{aligned}$$

$$f(\boldsymbol{\theta}) = \theta_1 \cos(\theta_2) + \theta_3 \sin(\theta_4)$$



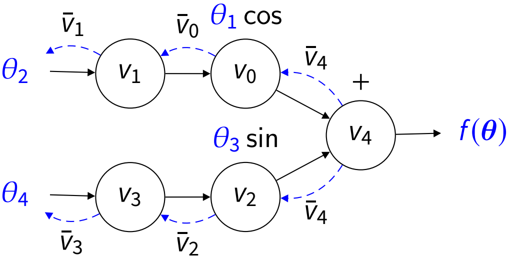
Reverse trace at $\boldsymbol{\theta} = \begin{bmatrix} 0.5 & 2 & 0.7 & 3 \end{bmatrix}$

$$\begin{aligned}
\bar{v}_5 &= \bar{f} && = 1 && \\
\bar{v}_4 &= \bar{v}_5 \; \partial v_5 / \partial v_4 && = 1 \cdot 1 && = 1 \\
\bar{v}_3 &= \bar{v}_5 \; \partial v_5 / \partial v_3 && = 1 \cdot 1 && = 1 \\
\bar{v}_2 &= \bar{v}_4 \; \partial v_4 / \partial v_2 && = 1 \cdot v_{-}1 && = 0.7 \\
\bar{v}_1 &= \bar{v}_3 \; \partial v_3 / \partial v_1 && = 1 \cdot v_{-}3 && = 0.5 \\
\bar{v}_0 &= \bar{v}_2 \; \partial v_2 / \partial v_0 && = 0.7 \cdot \cos(v_0) && = -0.693 \blacksquare \\
\bar{v}_{-1} &= \bar{v}_4 \; \partial v_4 / \partial v_{-1} && = 1 \cdot v_2 && = 0.141 \blacksquare \\
\bar{v}_{-2} &= \bar{v}_1 \; \partial v_1 / \partial v_{-2} && = 0.5 \cdot -\sin(v_{-2}) && = -0.454 \blacksquare \\
\bar{v}_{-3} &= \bar{v}_3 \; \partial v_3 / \partial v_{-3} && = 1 \cdot v_1 && = -0.416 \blacksquare
\end{aligned}$$

Can we save memory?

Embed multiplicative coefficients “inside” the node

- ○ The computational graph will lack some intermediate values
- ○ These values will have to be computed on the fly (by dividing with corresponding θ)



Primal trace at $\boldsymbol{\theta} = (0.5, 2, 0.7, 3)$

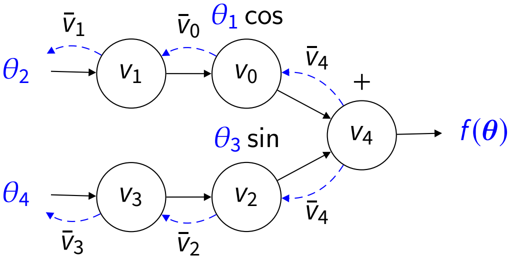
$$
\begin{aligned}
v_1 &= \theta_2 && = 2 \\
v_0 &= \theta_1 \cos(v_1) && = 0.5 \cos(2) && = -0.208 \\
v_3 &= \theta_4 && = 3 \\
v_2 &= \theta_3 \sin(v_3) && = 0.7 \sin(3) && = 0.099 \\
\frac{\partial f}{\partial \theta_1} &= \frac{\boldsymbol{\theta_1 \cos(v_1)}}{\boldsymbol{\theta_1}} && = -0.416 \blacksquare \\
\frac{\partial f}{\partial \theta_3} &= \frac{\boldsymbol{\theta_3 \sin(v_3)}}{\boldsymbol{\theta_3}} && = 0.141 \blacksquare
\end{aligned}
$$

- ○ Advantage: less memory, fewer nodes to visit
- ○ Disadvantage: intermediate values must be computed on the fly
- ○ Checkpointing: recompute data between checkpoints instead of storing in memory

Can we save memory?

Embed multiplicative coefficients “inside” the node

- The computational graph will lack some intermediate values
- These values will have to be computed on the fly (by dividing with corresponding θ)



Reverse trace at $\boldsymbol{\theta} = (0.5, 2, 0.7, 3)$

$$
\begin{aligned}
\bar{v}_4 &= \bar{f} && = 1 \\
\bar{v}_2 &= \bar{v}_4\, \partial v_4/\partial v_2 && = 1 \cdot 1 && = 1 \\
\bar{v}_0 &= \bar{v}_4\, \partial v_4/\partial v_0 && = 1 \cdot 1 && = 1 \\
\bar{v}_3 &= \bar{v}_2\, \partial v_2/\partial v_3 && = \theta_3 \cdot \frac{\boldsymbol{\theta_1 \cos(3)}}{\boldsymbol{\theta_1}} && = -0.693 \blacksquare \\
\bar{v}_1 &= \bar{v}_0\, \partial v_0/\partial v_1 && = \theta_1 \cdot \frac{\boldsymbol{-\theta_3 \sin(2)}}{\boldsymbol{\theta_3}} && = -0.454 \blacksquare
\end{aligned}
$$

- Advantage: less memory, fewer nodes to visit
- Disadvantage: intermediate values must be computed on the fly
- Checkpointing: recompute data between checkpoints instead of storing in memory

Implementation paradigms

- Elemental (replace math operations with calls to AD library)
- Operator overloading
  - Autodiff https://github.com/autodiff/autodiff
  - Ceres Solver https://ceres-solver.org
  - FADBAD++ http://uning.dk/fadbad.html
  - Stan Math https://github.com/stan-dev/math
- Compiler-based
  - Cla∂ https://github.com/vgvassilev/clad/
  - Enzyme https://enzyme.mit.edu/
  - Stalin∇ https://github.com/Functional-AutoDiff/STALINGRAD
  - DVL https://github.com/axch/dysvunctional-language
  - Tangent https://github.com/google/tangent
- Hybrid

Many approaches and optimizations are possible, see e.g. [Margossian, 2019]

Example implementation (C++)

Ingredients: tape + graph nodes + arithmetic “variable” with operator overloads

We assume only nullary, unary or binary variables

```
template<typename T>
concept Arithmetic = requires {
    std::is_arithmetic_v<T>;
};

template<Arithmetic T>
struct Node {
    std::array<T, 2> partials {T {0}, T {0}};  // partial derivative values
    std::array<std::size_t, 2> inputs {0, 0};  // indices of input nodes
};
```

```cpp
template<Arithmetic T> struct Var;  // forward declaration

template<Arithmetic T>
struct Tape {
    std::vector<Node<T>> nodes;

    auto push() -> std::size_t { ... }
    auto push(auto i, auto p) -> std::size_t { ... }
    auto push(auto i0, auto p0, auto i1, auto p1) -> std::size_t     {
        auto idx = std::size(nodes);
        nodes.push_back({{T {p0}, T {p1}}, {i0, i1}});
        return idx;
    }
    auto variable(T value) { return Var<T>{*this, value, push()}; }
    auto length() const -> std::size_t { return std::size(nodes); }
    auto clear() { nodes.clear(); }
}
```

```
template<Arithmetic T>
struct Var {
    Tape<T>& tape;  // reference to the tape
    std::size_t index {};  // index of the current node
    T value {};  // associated value

    explicit Var(Tape<T>& t, T v = T {0}, std::size_t i = 0)
        : tape(t), index(i), value(v) { }

    auto gradient() const -> grad<T> {
        std::vector<T> grad(tape.length(), T {0.0});
        grad[index] = 1.0;
        for (auto i = tape.length() - 1; i < tape.length(); --i) {
            auto const& n = tape.nodes[i];
            for (auto j = 0UL; j < std::size(n.inputs); ++j) {
                grad[n.inputs[j]] += n.partials[j] * grad[i];
            }
        }
        return {grad};
    }
    // + a bunch of operator overloads
}
```

```
template<Arithmetic T>
struct Var {
    // ... continued (1)

    friend auto operator+(Var const& a, Var const& b) -> Var {
        return Var {a.tape, a.value + b.value,
                    a.tape.push(a.index, T {1.0}, b.index, T {1.0})};
    }
    friend auto operator-(Var const& a, Var const& b) -> Var {
        return Var {a.tape, a.value - b.value,
                    a.tape.push(a.index, T {1.0}, b.index, T {-1.0})};
    }
    friend auto operator*(Var const& a, Var const& b) -> Var {
        return Var {a.tape, a.value * b.value,
                    a.tape.push(a.index, b.value, b.index, a.value)};
    }
    friend auto operator/(Var const& a, Var const& b) -> Var {
        return Var {a.tape, a.value / b.value,
                    a.tape.push(a.index, 1 / b.value, b.index,
                                -a.value / (b.value * b.value))};
    }
}
```

```cpp
template<Arithmetic T>
struct Var {
    // ... continued (2)

    auto sin() const -> Var {
        return Var {tape, std::sin(value), tape.push(index, std::cos(value))};
    }

    auto cos() const -> Var {
        return Var {tape, std::cos(value), tape.push(index, -std::sin(value))};
    }

    auto exp() const -> Var {
        return Var {tape, std::exp(value), tape.push(index, std::exp(value))};
    }

    auto log() const -> Var {
        return Var {tape, std::log(value), tape.push(index, 1 / value)};
    }
}
```

$$f(x,y) = \ln(x) + xy - \sin(y)$$
$$\nabla f_{|x=2,y=5} = [5.5, 1.716]$$

```cpp
"log(x) + xy - sin(y) | x=2, y=5"_test = [&]
{
    auto constexpr a {2.0};
    auto constexpr b {5.0};

    Tape<double> tape;
    auto x = tape.variable(a);
    auto y = tape.variable(b);
    auto z = x.log() + x * y - y.sin();
    auto g = z.gradient();

    expect(eq(z.value, std::log(a) + a * b - std::sin(b)));
    expect(eq(g.wrt(x), b + 1 / a));
    expect(eq(g.wrt(y), a - std::cos(b)));
};
```

(boost::μt https://boost-ext.github.io/ut/)

$$f(x, y) = \ln(x) + xy - \sin(y)$$
$$\nabla f_{|x=2, y=5} = [5.5, 1.716]$$

```python
from sympy import diff, sin, cos, tan, exp, log
from sympy import Matrix, Symbol

x = Symbol('x', real=True)
y = Symbol('y', real=True)

F = log(x) + x * y - sin(y)
J = Matrix([[ diff(F, x), diff(F, y )]])
J.subs([(x, 2.0), (y, 5.0)]) # [ 5.5 1.71633781453677 ]
```

(SymPy https://www.sympy.org/en/index.html)

Nonlinear Least Squares

Let $x \in \mathbb{R}^n$ and $F(x) = \left[f_1(x), \ldots, f_m(x)\right]^T \in \mathbb{R}^m$

We want to solve the optimization problem

$$\arg\min_x \frac{1}{2}\|F(x)\|^2$$

Let $J(x) \in \mathbb{R}^{m \times n}$, $J_{ij}(x) = \partial_j f_i(x)$ and $g(x) = \nabla \frac{1}{2}\|F(x)\|^2 = J(x)^T F(x)$

Then we can use

$$F(x + \Delta x) \approx F(x) + J(x)\Delta x$$

which leads to the linear least squares problem

$$\min_{\Delta x} = \frac{1}{2}\|J(x)\Delta x + F(x)\|^2$$

In order to converge, step size Δx must be adapted [Nocedal and Wright, 2006]

Nonlinear Least Squares

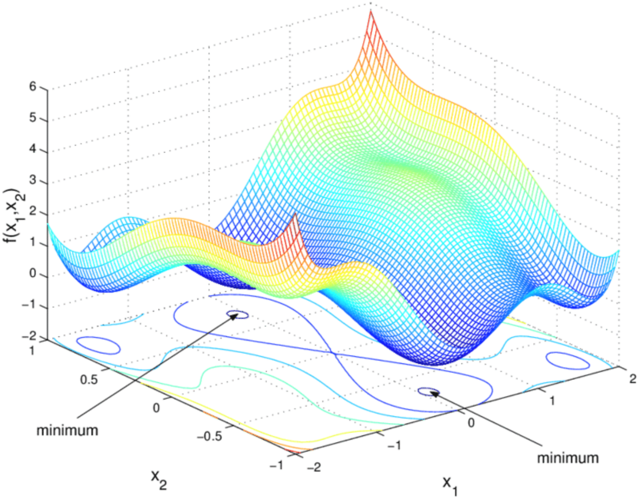


Image source: [Klerk et al., 2014]

NIST – Thurber Dataset – Semiconductor electron mobility modeling

https://www.itl.nist.gov/div898/strd/nls/data/thurber.shtml

$$y = f(x;\beta) + \epsilon = \frac{\beta_1 + \beta_2 x + \beta_3 x^2 + \beta_4 x_3}{1 + \beta_5 x + \beta_6 x^2 + \beta_7 x^3} + \epsilon$$

```
                    Certified               Certified
Parameter           Estimate                Std. Dev. of Est.
beta(1)             1.2881396800E+03        4.6647963344E+00
beta(2)             1.4910792535E+03        3.9571156086E+01
beta(3)             5.8323836877E+02        2.8698696102E+01
beta(4)             7.5416644291E+01        5.5675370270E+00
beta(5)             9.6629502864E-01        3.1333340687E-02
beta(6)             3.9797285797E-01        1.4984928198E-02
beta(7)             4.9727297349E-02        6.5842344623E-03
Residual
Sum of Squares      5.6427082397E+03
Standard Deviation  1.3714600784E+01
Degrees of Freedom  30
```

```cpp
struct thurber_functor {
    using Scalar = double;
    using JacobianType = Eigen::Matrix<Scalar, -1, -1>;
    using QRSolver = Eigen::ColPivHouseholderQR<JacobianType>;
    static constexpr std::array start1 { /* predefined start */ };
    static constexpr std::array start2 { /* another predefined start */ };
    static constexpr std::array xval { /* dataset values */ };
    static constexpr std::array yval { /* dataset values */ };
    [[nodiscard]] auto values() const -> int { return xval.size(); }
    [[nodiscard]] auto inputs() const -> int { return start1.size(); }

    auto operator()(Eigen::Matrix<Scalar, -1, 1> const& input,
                    Eigen::Matrix<Scalar, -1, 1>& residual) const -> int {
        return (*this)(input, residual.data(), static_cast<Scalar*>(nullptr));
    }

    auto df(Eigen::Matrix<Scalar, -1, 1> const& input,
            Eigen::Matrix<Scalar, -1, -1>& jacobian) const -> int {
        return (*this)(input, static_cast<Scalar*>(nullptr), jacobian.data());
    }
    // to be continued
};
```

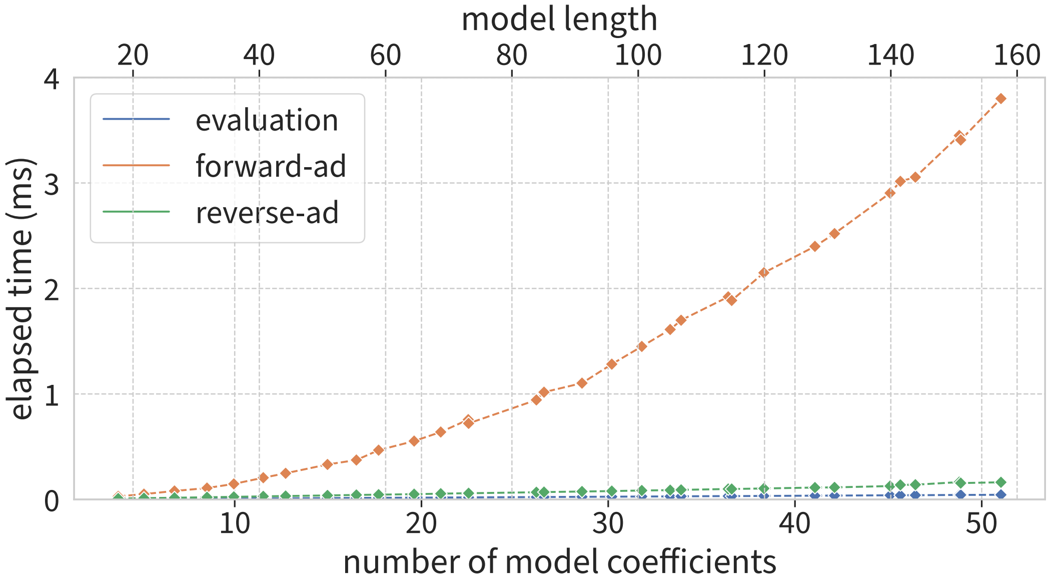
```
auto operator()(auto const& input, auto* residual, auto* jacobian) const -> int {
    reverse::Tape<Scalar> tape;
    std::vector<decltype(tape)::Variable> beta;
    for (auto i = 0; i < std::ssize(xval); ++i) {
        tape.clear(); beta.clear();
        for (auto v : input) { beta.push_back(tape.variable(v)); }

        auto x = xval[i], xx = x * x, xxx = x * x * x;
        auto f = (beta[0] + beta[1] * x + beta[2] * xx + beta[3] * xxx) /
                 (1 + beta[4] * x + beta[5] * xx + beta[6] * xxx);

        if (residual != nullptr) { residual[i] = f.value - yval[i]; }
        if (jacobian != nullptr) {
            auto g = f.gradient();
            for (auto const& b : beta) {
                jacobian[values() * b.index + i] = g.wrt(b);
            }
        }
    }
    return 0;
}
```

```
"thurber"_test = [&]
{
    auto constexpr tol {1.E4 * std::numeric_limits<double>::epsilon()};
    auto constexpr max_fun_eval {50};

    auto s1 = thurber_functor::start1;  // try first starting point
    Eigen::VectorXd x = Eigen::Map<decltype(x) const>(s1.data(), std::ssize(s1));

    //https://eigen.tuxfamily.org/dox/unsupported/classEigen_1_1LevenbergMarquardt.html
    thurber_functor cost_function;
    Eigen::LevenbergMarquardt<thurber_functor> lm(cost_function);
    Eigen::LevenbergMarquardtSpace::Status status {};
    lm.setMaxfev(max_fun_eval);
    lm.setFtol(tol);
    lm.setXtol(tol);
    status = lm.minimize(x);

    auto constexpr expected_norm {5.6427082397E+03};
    auto constexpr eps {1e-4};
    expect(approximately_equal {eps}(lm.fvec().squaredNorm(), expected_norm));
}
```

```
Running test "thurber"...
iteration         1        cost        664958
iteration         2        cost        35841.2
iteration         3        cost        13750.3
iteration         4        cost        13464.4
iteration         5        cost        7812.71
iteration         6        cost        7167.11
iteration         7        cost        6338.15
iteration         8        cost        6092.02
iteration         9        cost        5728.47
iteration        10        cost        5683.7
iteration        11        cost        5649
iteration        12        cost        5646.6
iteration        13        cost        5643.99
iteration        14        cost        5643.4
iteration        15        cost        5642.98
iteration        16        cost        5642.84
iteration        17        cost        5642.77
iteration        18        cost        5642.74
iteration        19        cost        5642.72
iteration        20        cost        5642.71
```

Operon Autodiff Module

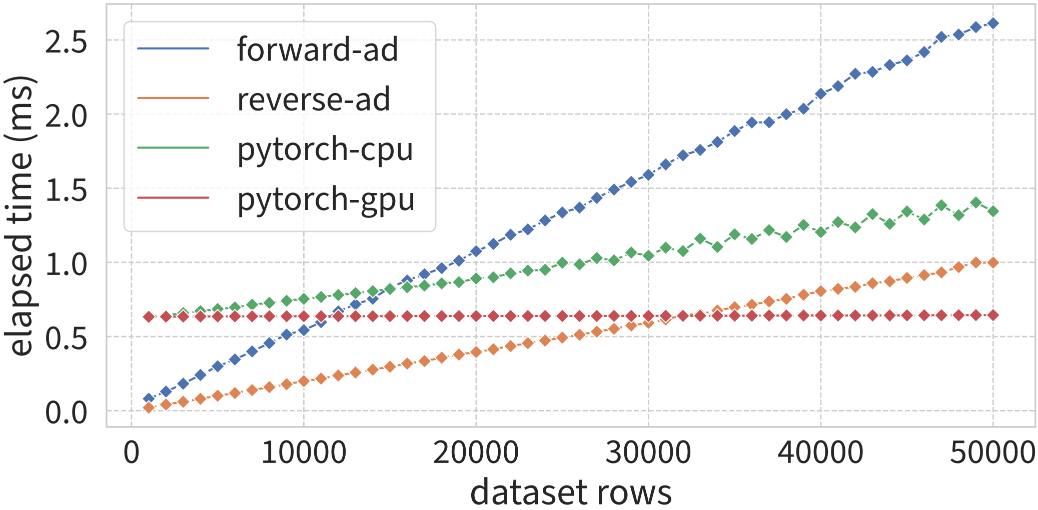
`https://github.com/heal-research/operon/tree/cpp20/include/operon/autodiff`


model length
20
40
60
80
100
120
140
160
4
3
2
1
0
elapsed time (ms)
evaluation
forward-ad
reverse-ad
10
20
30
40
50
number of model coefficients

CPU: 5950X

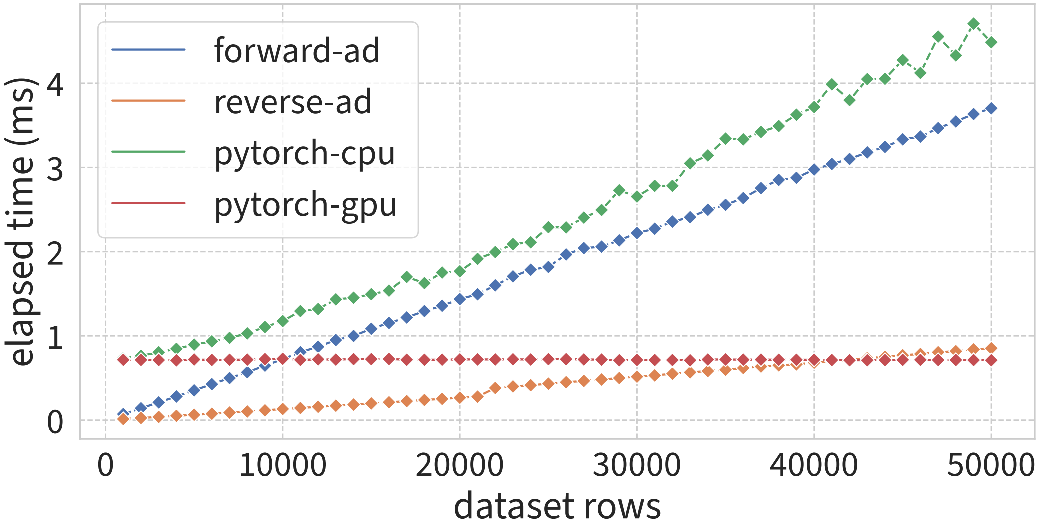
```
import torch
from functorch import jacrev, vmap

import numpy as np
import timeit

torch.set_default_dtype(torch.float32)
torch.set_num_threads(1)

def F1(X):
    return 10 * (math.pi * X[0] * X[1]).sin() \
              + 20 * (X[2] - 0.5) ** 2 \
              + 10*X[3] + 5*X[4]

nrow, ncol = 50000, 5
X = torch.rand(nrow, ncol)

for rows in np.arange(1000, nrow+1, 1000):
    %timeit -n 100 j = vmap(jacrev(F1))(X[:rows,:])
```

$$F_1 = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5$$



CPU: 5950X, GPU: 2080S

$$F_2 = x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$$



CPU: 5950X, GPU: 2080S

Conclusion

Programming backprop using hand-written derivatives is like coding in assembly

Using AD, it is also possible to differentiate loops and conditional statements

Reverse AD: Relatively simple to implement, hard to implement *efficiently*

The cheap gradient principle and its implications not commonly understood

Full Jacobians and Hessians may be an order of magnitude more expensive

Differentiable programming becoming increasingly popular
`https://diffprogramming.mit.edu/`

Thank you

F. L. Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974. doi: 10.1137/0711010. URL https://doi.org/10.1137/0711010.

Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, jan 2017. ISSN 1532-4435.

B. Christianson. *A Leibniz notation for automatic differentiation*, pages 1–9. Lecture Notes in Computational Science and Engineering. Springer, 2012. ISBN 978-3-642-30022-6. doi: 10.1007/978-3-642-30023-3_1.

Clifford. Preliminary sketch of biquaternions. *Proceedings of The London Mathematical Society*, pages 381–395, 1871.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, USA, second edition, 2008. ISBN 0898716594.

Hecht-Nielsen. Theory of the backpropagation neural network. In *International 1989 Joint Conference on Neural Networks*, pages 593–605 vol.1, 1989. doi: 10.1109/IJCNN.1989.118638.

Etienne Klerk, Cornelis Roos, and Tamás Terlaky. *Nonlinear Optimization text-book*. 03 2014.

Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery*, 9(4), mar 2019. doi: $10.1002/\mathrm{widm}.1305$. URL https://doi.org/10.1002%2Fwidm.1305.

J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN 9780387400655. URL https://books.google.at/books?id=VbHYoSyelFcC.

D.B. Parker. *Learning-logic: Casting the Cortex of the Human Brain in Silicon*. Technical report: Center for Computational Research in Economics and Management Science. Massachusetts Institute of Technology, 1985. URL https://books.google.at/books?id=2kS9GwAACAAJ.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

Andrea Walther. Getting started with adol-c. In *Combinatorial Scientific Computing*, 2009.

Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341700. URL https://doi.org/10.1145/3341700.

R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8): 463–464, aug 1964. ISSN 0001-0782. doi: 10.1145/355586.364791. URL https://doi.org/10.1145/355586.364791.

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