Introduction to Automatic Differentiation

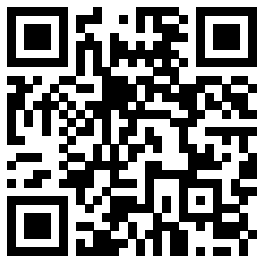
Bogdan Burlacu

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

Automatic differentiation

Arguably, the subject began with Newton and Leibniz.

...advances by 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).

Stigler's Law of Eponymy

No scientific law is named after its original discoverer.

Phil Wolfe, 1982

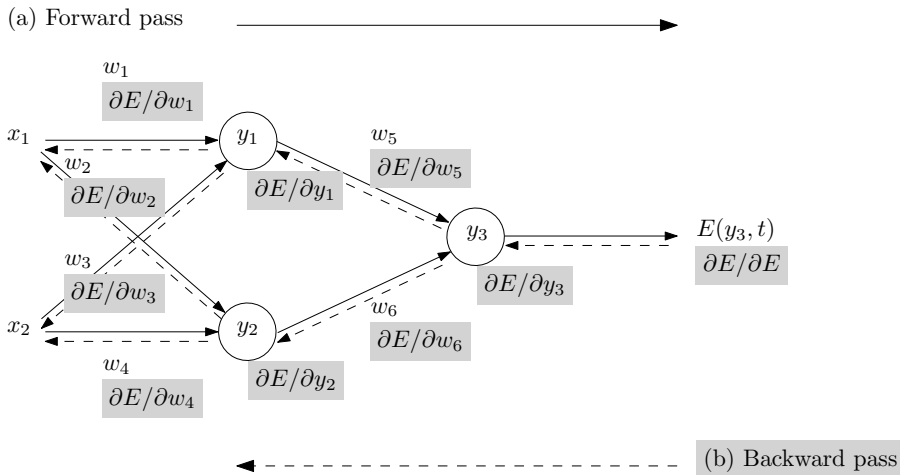
There is a common misconception that calculating a function of n variables and its gradient is about $(n + 1)$ times as expensive as just calculating the function [...] If care is taken in handling quantities, which are common to the function and its derivatives, the ratio is usually 1.5, not $(n + 1)$.

Cheap gradient principle

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

There is no cheap Jacobian principle

Gradient-based optimization is a pillar of machine learning



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

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*
   - Simple but inexact
   - Complexity linear in the number of variables

Methods

1. *Analytical*
   - Work out derivatives by hand
   - Time-consuming and error prone
2. *Numerical*
   - Simple but inexact
   - 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*
   - Simple but inexact
   - 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*
   - Function is decomposed into sequence of elementary operations
   - Relies on the application of the chain rule of calculus
   - *Exact* (up to machine precision)

Chain rule

If f is a composite function

$$g : \mathbb{R}^k \to \mathbb{R}^m$$
$$h : \mathbb{R}^n \to \mathbb{R}^k$$
$$f : \mathbb{R}^n \to \mathbb{R}^m$$
$$f = g \circ h = g\left(h(x)\right)$$

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 $2 \times 4 = 8$ 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]

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

Reverse mode

The expression of a function to be differentiated can be sorted into a *topological graph*

Each node represents one elementary operation such as $\div$ or $\log$ (division or logarithm)

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:

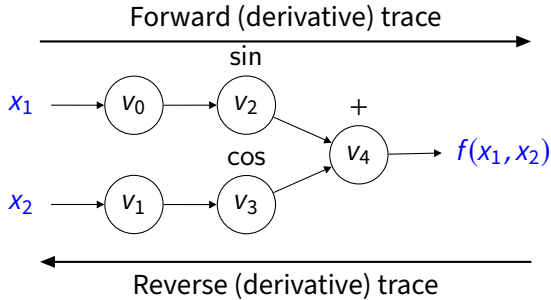
- $n \cdot c \cdot \text{ops}(f)$ in forward mode
- $m \cdot c \cdot \text{ops}(f)$ in reverse mode

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

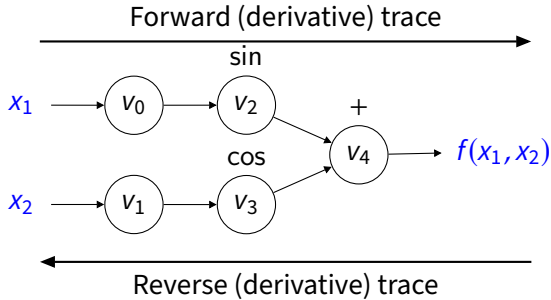
Computational graph [Bauer, 1974]

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



Computational graph [Bauer, 1974]

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



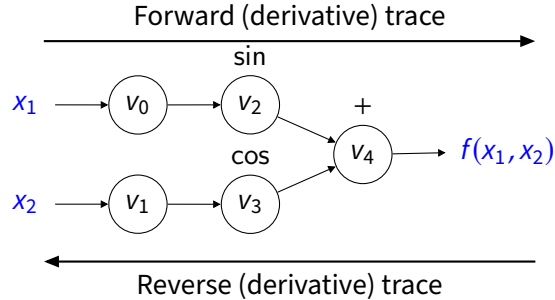
Topology

$$v_0 \prec v_2 \prec v_4$$

$$v_1 \prec v_3 \prec v_4$$

Computational graph [Bauer, 1974]

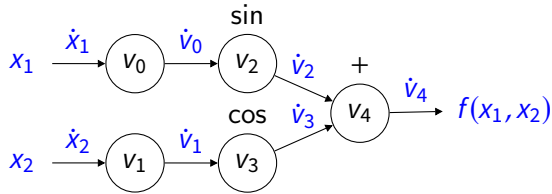
$f(x_1, x_2) = \sin(x_1) + \cos(x_2)$



Evaluation at $(x_1, x_2) = (2, 3)$

v_0	=	x_1	=	2		
v_1	=	x_2	=	3		
v_2	=	$\sin(v_0)$	=	$\sin(2)$	=	0.909
v_3	=	$\cos(v_1)$	=	$\cos(3)$	=	−0.990
v_4	=	$v_2 + v_3$	=	$0.909 - 0.990$	=	−0.081

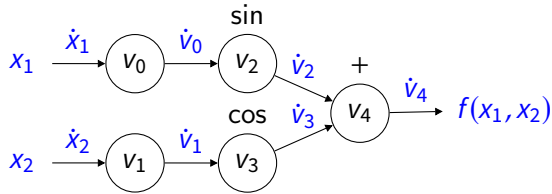
$f(x_1, x_2) = \sin(x_1) + \cos(x_2)$



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

$$
\begin{array}{lcllcl}
\dot{v}_0 & = & \dot{x}_1 & = & 1 & \\
\dot{v}_1 & = & \dot{x}_2 & = & 0 & \\
\dot{v}_2 & = & \dot{v}_0 \cdot \cos(v_0) & = & 1 \cdot \cos(2) & = & -0.416 \\
\dot{v}_3 & = & \dot{v}_1 \cdot -\sin(v_1) & = & 0 \cdot -\sin(3) & = & 0 \\
\dot{v}_4 & = & \dot{v}_2 + \dot{v}_3 & = & -0.416 + 0 & = & -0.416 \blacksquare
\end{array}
$$

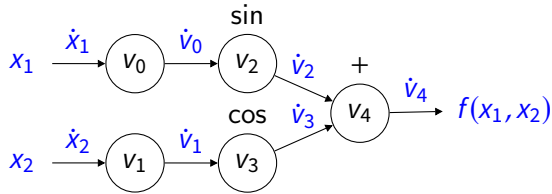
$f(x_1, x_2) = \sin(x_1) + \cos(x_2)$



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

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

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

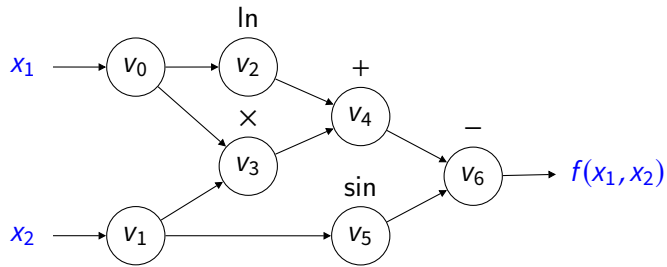


Reverse trace at $(x_1, x_2) = (2, 3)$ for $\dot{x}_2$

$$
\begin{array}{lcllcl}
\bar{v}_4 & = & 1 & & & \\
\bar{v}_3 & = & \bar{v}_4 \, \partial v_4 / \partial v_3 & = & 1 \cdot 1 & = & 1 \\
\bar{v}_2 & = & \bar{v}_4 \, \partial v_4 / \partial v_2 & = & 1 \cdot 1 & = & 1 \\
\bar{v}_1 & = & \bar{v}_3 \, \partial v_3 / \partial v_1 & = & 1 \cdot -\sin(3) & = & -0.141 \blacksquare \\
\bar{v}_0 & = & \bar{v}_2 \, \partial v_2 / \partial v_0 & = & 1 \cdot \cos(2) & = & -0.416 \blacksquare
\end{array}
$$

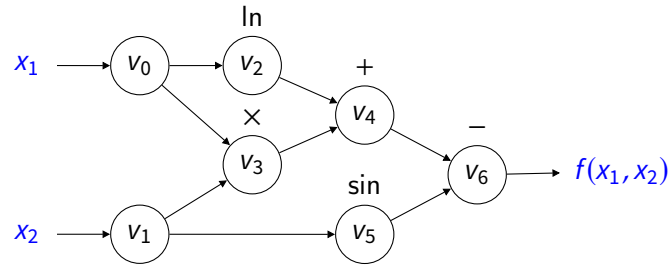
Computational graph

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



[Baydin et al., 2017]

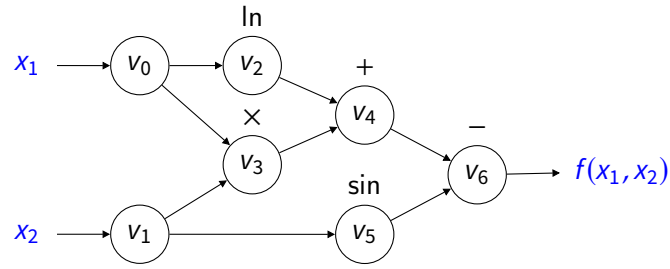
$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



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

v_0	$=$	x_1	$=$	2		
v_1	$=$	x_2	$=$	5		
v_2	$=$	$\ln(v_1)$	$=$	$\ln(2)$	$=$	0.693
v_3	$=$	$x_1 x_2$	$=$	$2 \cdot 5$	$=$	10
v_4	$=$	$v_2 + v_3$	$=$	$0.693 + 10$	$=$	10.693
v_5	$=$	$\sin(v_1)$	$=$	$\sin(5)$	$=$	-0.959
v_6	$=$	$v_4 - v_5$	$=$	$10.693 + 0.959$	$=$	11.652

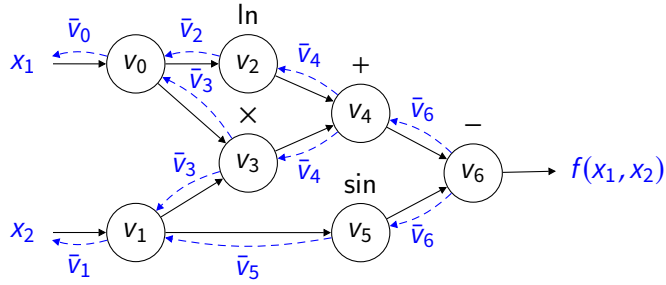
$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



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

$$
\begin{array}{lllllll}
\dot{v}_0 & = & \dot{x}_1 & = & 1 & & \\
\dot{v}_1 & = & \dot{x}_2 & = & 0 & & \\
\dot{v}_2 & = & \dot{v}_0 / v_0 & = & 0.5 & & \\
\dot{v}_3 & = & \dot{v}_0 v_1 + \dot{v}_1 v_0 & = & 1 \cdot 5 + 0 \cdot 2 & = & 5 \\
\dot{v}_4 & = & \dot{v}_2 + \dot{v}_3 & = & 0.5 + 5 & = & 5.5 \\
\dot{v}_5 & = & \dot{v}_1 \cdot \cos(v_5) & = & 0 \cdot \cos(5) & = & 0 \\
\dot{v}_6 & = & \dot{v}_4 - \dot{v}_5 & = & 5.5 - 0 & = & 5.5 \ \blacksquare
\end{array}
$$

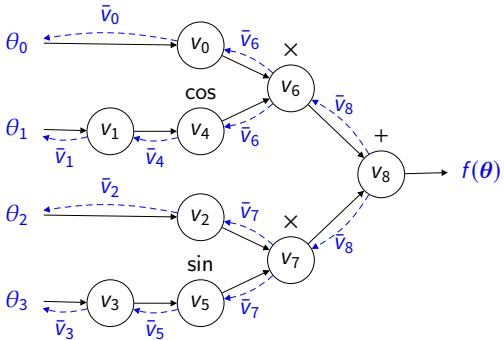
$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



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

$$
\begin{array}{lcllcl}
\bar{v}_6 & = & \bar{f} & = & 1 & \\
\bar{v}_5 & = & \bar{v}_6 \, \partial v_6 / \partial v_5 & = & 1 \cdot (-1) & = & -1 \\
\bar{v}_4 & = & \bar{v}_6 \, \partial v_6 / \partial v_4 & = & 1 \cdot 1 & = & 1 \\
\bar{v}_3 & = & \bar{v}_4 \, \partial v_4 / \partial v_3 & = & 1 \cdot 1 & = & 1 \\
\bar{v}_2 & = & \bar{v}_4 \, \partial v_4 / \partial v_2 & = & 1 \cdot 1 & = & 1 \\
\bar{v}_1 & = & \bar{v}_3 \, \partial v_3 / \partial v_1 + \bar{v}_5 \, \partial v_5 / \partial v_1 & = & v_0 - \cos(v_1) & = & 1.716 \, \blacksquare \\
\bar{v}_0 & = & \bar{v}_3 \, \partial v_3 / \partial v_1 + \bar{v}_2 \, \partial v_2 / \partial v_1 & = & v_1 + 1/v_0 & = & 5.5 \, \blacksquare
\end{array}
$$

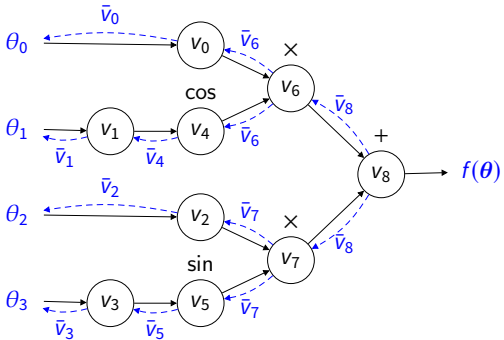
$$f(\boldsymbol{\theta}) = \theta_0 \cos(\theta_1) + \theta_2 \sin(\theta_3)$$



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

$$
\begin{aligned}
v_0 &= \theta_0 &&= 0.5 \\
v_1 &= \theta_1 &&= 2 \\
v_2 &= \theta_2 &&= 0.7 \\
v_3 &= \theta_3 &&= 3 \\
v_4 &= \cos(v_1) &&= -0.416 \\
v_5 &= \sin(v_3) &&= 0.141 \\
v_6 &= v_0 \cdot v_4 &&= -0.208 \\
v_7 &= v_2 \cdot v_5 &&= 0.099 \\
v_8 &= v_6 + v_7 &&= -0.109
\end{aligned}
$$

$$f(\boldsymbol{\theta}) = \theta_0 \cos(\theta_1) + \theta_2 \sin(\theta_3)$$



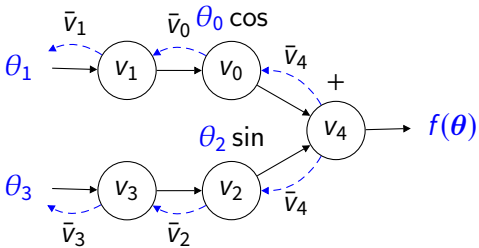
Reverse trace at $\boldsymbol{\theta} = (0.5, 2, 0.7, 3)$

$$
\begin{aligned}
\bar{v}_8 &= \bar{f} && = 1 \\
\bar{v}_7 &= \bar{v}_8\, \partial v_8/\partial v_7 && = 1 \cdot 1 && = 1 \\
\bar{v}_6 &= \bar{v}_8\, \partial v_8/\partial v_6 && = 1 \cdot 1 && = 1 \\
\bar{v}_5 &= \bar{v}_7\, \partial v_7/\partial v_5 && = 1 \cdot v_2 && = 0.7 \\
\bar{v}_4 &= \bar{v}_6\, \partial v_6/\partial v_4 && = 1 \cdot v_0 && = 0.5 \\
\bar{v}_3 &= \bar{v}_5\, \partial v_5/\partial v_3 && = 0.7 \cdot \cos(v_3) && = -0.693 \blacksquare \\
\bar{v}_2 &= \bar{v}_7\, \partial v_7/\partial v_2 && = 1 \cdot v_5 && = 0.141 \blacksquare \\
\bar{v}_1 &= \bar{v}_4\, \partial v_4/\partial v_1 && = 0.5 \cdot -\sin(v_1) && = -0.454 \blacksquare \\
\bar{v}_0 &= \bar{v}_6\, \partial v_6/\partial v_0 && = 1 \cdot v_4 && = -0.416 \blacksquare
\end{aligned}
$$

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)



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

$$
\begin{aligned}
v_1 &= \theta_1 &&= 2 \\
v_0 &= \theta_0 \cos(v_1) &&= 0.5\cos(2) = -0.208 \\
v_3 &= \theta_3 &&= 3 \\
v_2 &= \theta_2 \sin(v_3) &&= 0.7\sin(3) = 0.099
\end{aligned}
$$

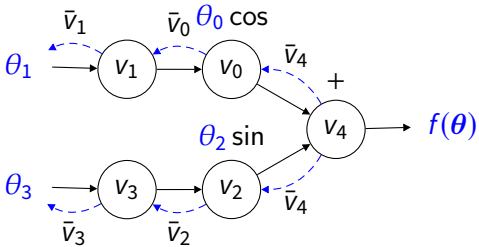
$$
\begin{aligned}
\frac{\partial f}{\partial \theta_0} &= \frac{\boldsymbol{\theta_0 \cos(v_1)}}{\boldsymbol{\theta_0}} &&= -0.416 \blacksquare \\
\frac{\partial f}{\partial \theta_2} &= \frac{\boldsymbol{\theta_2 \sin(v_3)}}{\boldsymbol{\theta_2}} &&= 0.141 \blacksquare
\end{aligned}
$$

- Advantage: less memory, fewer nodes to visit
- Disadvantage: need to divide for intermediate quantities

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

$$\bar{v}_4 = \bar{f} = 1$$

$$\bar{v}_2 = \bar{v}_4\, \partial v_4/\partial v_2 = 1 \cdot 1 = 1$$

$$\bar{v}_0 = \bar{v}_4\, \partial v_4/\partial v_0 = 1 \cdot 1 = 1$$

$$\bar{v}_3 = \bar{v}_2\, \partial v_2/\partial v_3 = \theta_2 \cdot \frac{\theta_0 \cos(3)}{\theta_0} = -0.693 \blacksquare$$

$$\bar{v}_1 = \bar{v}_0\, \partial v_0/\partial v_1 = \theta_0 \cdot \frac{-\theta_2 \sin(2)}{\theta_2} = -0.454 \blacksquare$$

- Advantage: less memory, fewer nodes to visit
- Disadvantage: need to divide for intermediate quantities

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`
- 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`
- Hybrid

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

Reverse AD Implementation (C++20)

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(); }
}
```

```cpp
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
}
```

```cpp
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)};
    }
}
```

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

```
"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/)

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

```
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)

NIST – Thurber Dataset – Semiconductor electron mobility modeling

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

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

Parameter	Certified Estimate	Certified 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	

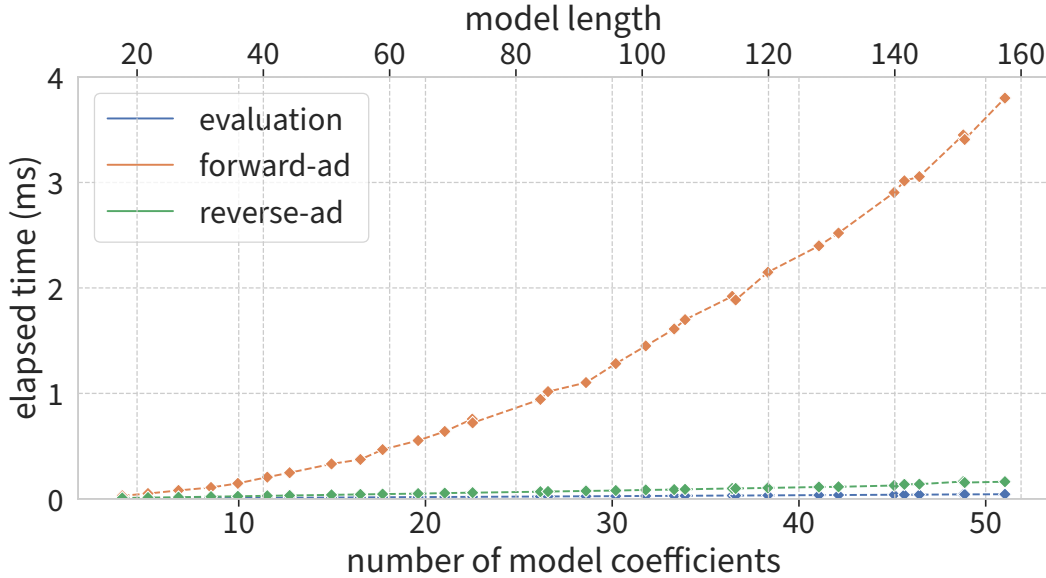
```cpp
struct thurber_functor {
    using Scalar = double;
    using JacobianType = Eigen::Matrix<Scalar, -1, -1>;
    using QRSolver = Eigen::ColPivHouseholderQR<JacobianType>;
    static constexpr std::array start1 { /* predefined start */ };
    static constexpr std::array start2 { /* another predefined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

    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;
}
```

```cpp
"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));
}
```

Operon Autodiff Module

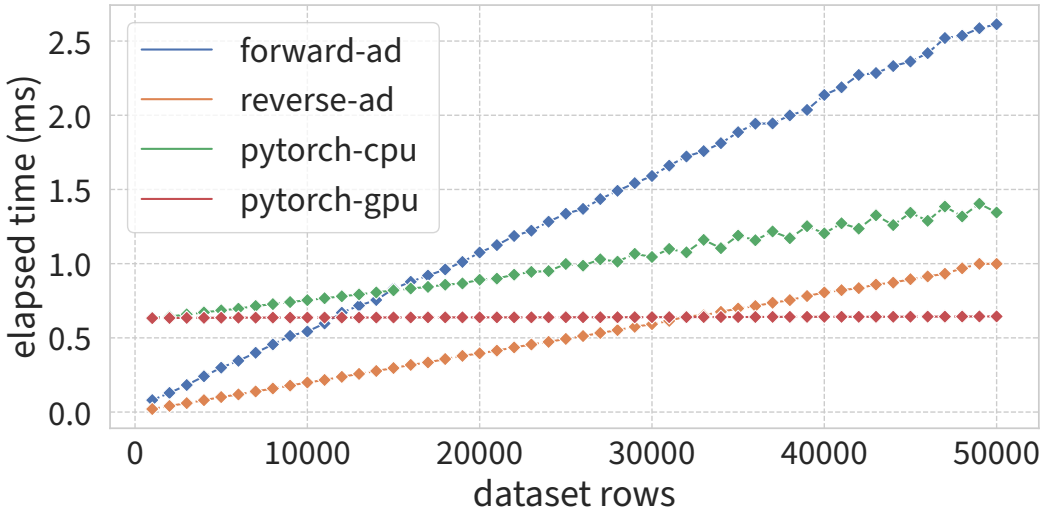
https://github.com/heal-research/operon/tree/cpp20/include/operon/autodiff


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

CPU: 5950X

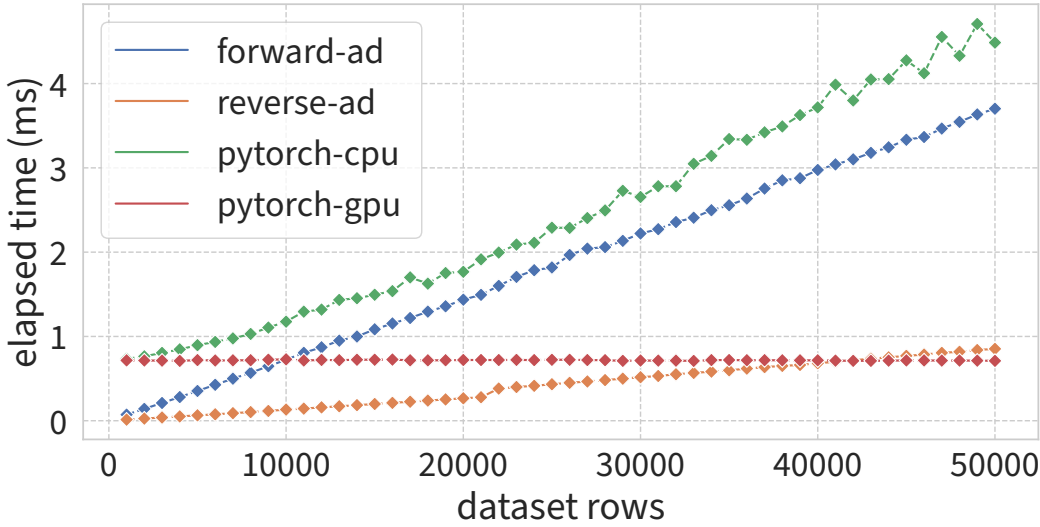
```
import torch
from functorch import jacrev, vmap

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

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



GPU: 2080S, CPU: 5950X

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

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.

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.

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`.