Diferențierea Automată în Machine Learning

Fundamentele Matematice ale Inteligenței Artificiale

DI Dr. Bogdan Burlacu
Universitatea Tehnică Gheorghe Asachi Iași
Facultatea de Automatică și Calculatoare

Introducere

Ce este diferențierea automată

Metodă pentru calculul exact, complet automat, al gradienților.

- programul este văzut ca un graf de expresii elementare
- se aplică formula de derivare a funcțiilor compuse (“chain rule”) pe acel graf

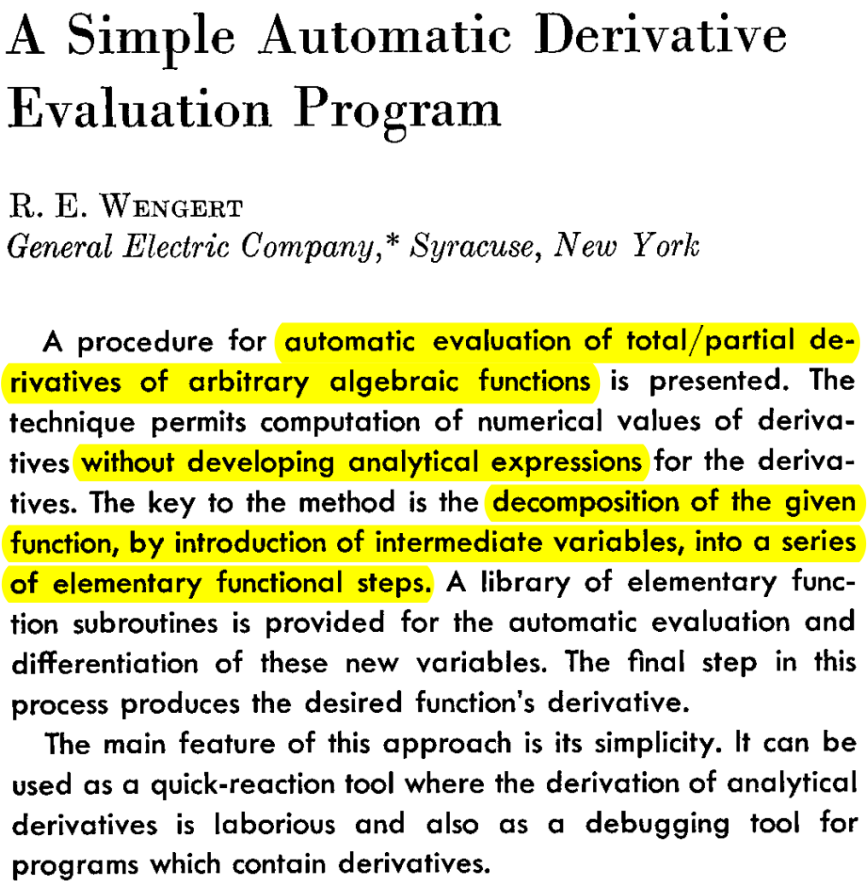
Avantaje

- precizie limitată doar de capacitatea de reprezentare în floating-point
- permite diferențierea fluxului de control (bucle `while`, `for` de lungime arbitrară)

Aplicații

Orice metodă de optimizare bazată pe gradient, mai ales în Deep Learning.

Introducere



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.

Figure 1: R. E. Wengert [1]

Calcul diferențial

Subiectul a început de la Newton și Leibniz

> 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

Diferențiere automată

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 and 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).

Diferențiere automată

Subiectul a început cu Newton și Leibniz

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

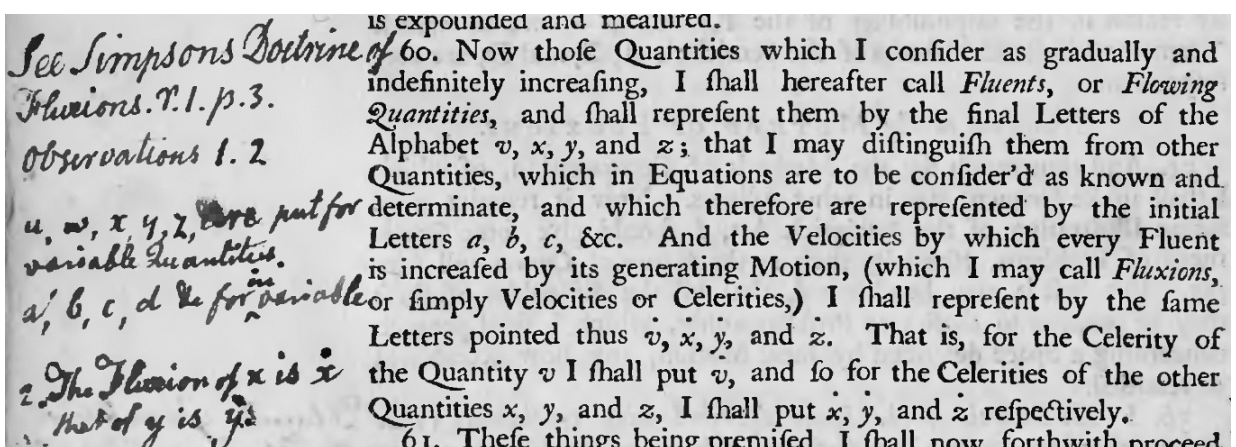


Figure 2: I. Newton

Diferențiere automată

...cu contribuții de la mulți alții

> *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)*

În special, Andreas Griewank este considerat un personaj central în "renașterea modernă" a diferențierii automate (în principal datorită cercetării pe tema "reverse mode AD").

În această perioadă, algorithmul backpropagation a fost reinventat de mai multe ori în comunitatea de învățare automată D. Parker [2], D. E. Rumelhart, G. E. Hinton, and R. J. Williams [3], P. J. Werbos [4]

Teoria a fost unificată mai târziu de către Hecht-Nielsen [5].

Diferențiere automată

Stigler's Law of Eponimy

No scientific discovery is named after its original discoverer.

- Hubble's law, derived by Georges Lemaître two years before Edwin Hubble
- Pythagorean theorem, known to Babylonian mathematicians before Pythagoras

Principiul gradientului ieftin

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

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

Prima aplicație la scară largă a optimizării bazată pe gradient a fost în teoria controlului.

Nu există nici un principiu al “Jacobianului ieftin”.

Optimizarea bazată pe gradient

Reprezintă baza fundamentală a învățării automate.

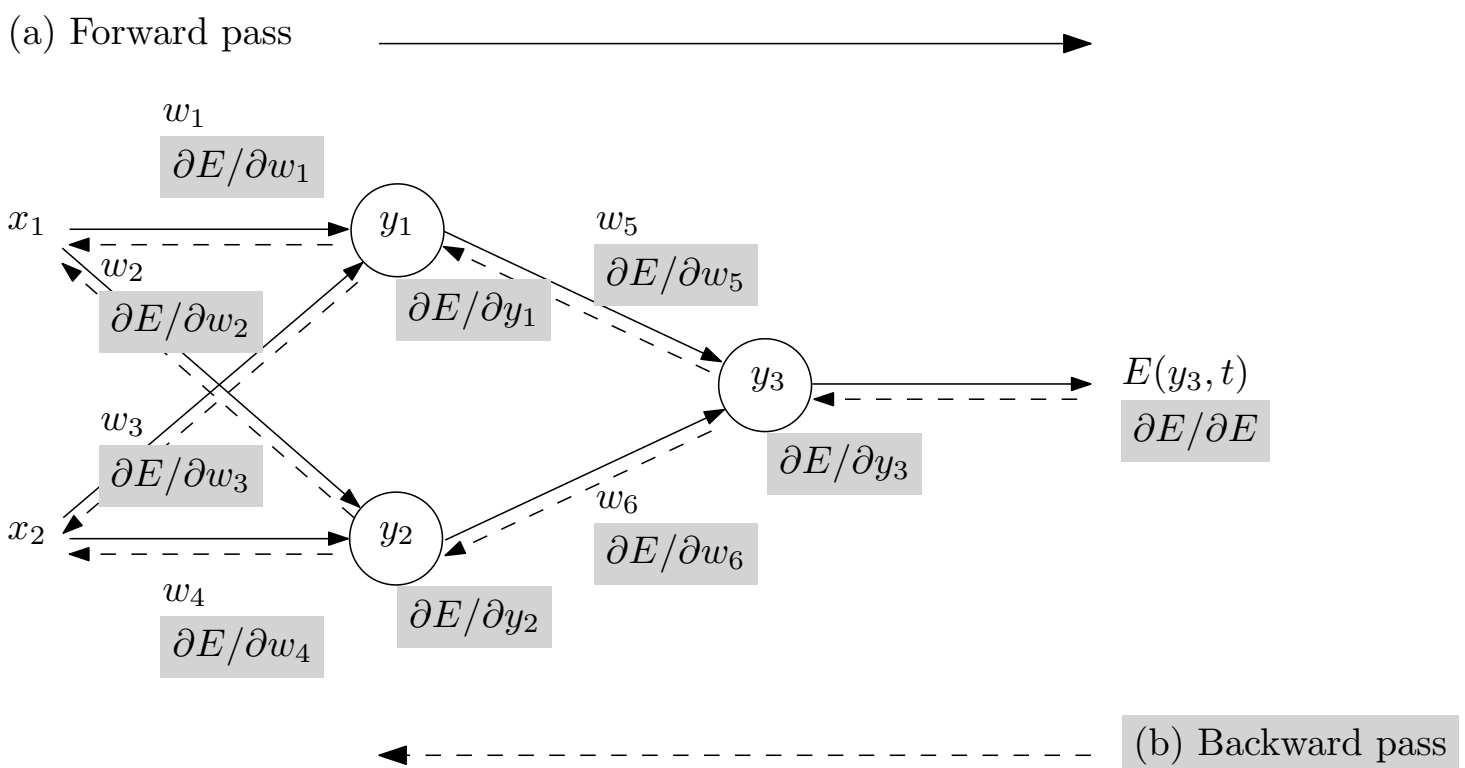


Figure 3: A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind [6]

Domenii de aplicare

- Chimie atmosferică
- Finanțe computaționale
- Mecanica fluidelor numerică (computational fluid dynamics)
- Ray tracing diferențiabil
- Inginerie (design și optimizare)
- Învățare automată

Deep learning → differentiable programming

- ANN-uri asamblate din blocuri de bază și antrenate prin backpropagation
- Elemente algoritmice *continue* și *diferențiabile*
- În arhitecturi complexe, diferențiabilitatea înseamnă că pot fi antrenate "end-to-end"
  F. Wang, D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf [7]
- Suport pentru diferențiere automată la nivel de limbaj și compilator

Baze teoretice

Având o funcție

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

cu matricea Jacobiană

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

J_f exprimă rata de schimbare a fiecărei componente a lui f coresponzătoare fiecărei componente a lui $x \in \mathbb{R}^n$.

Metode de calcul

1. Analitic
   - calcul manual al derivatelor
   - susceptibil la erori, costisitor
2. Numeric
   - utilizează definiția la limită a derivatei
   - metodă simplă dar inexactă, susceptibilă la erori de aproximare
   - complexitate liniară în numărul de variabile
3. Simbolic
   - manipulare simbolică a expresiilor în formă închisă
   - expresii complexe, criptice, susceptibile la explozie în dimensiune
4. Automată (*algoritmică*)

Diferențierea automată

Ar putea fi numită mai exact *diferențiere algoritmică*:

- funcția este descompusă în secvența de operații elementare
- bazată pe formula de derivare a funcțiilor compuse
- păstrează capacitatea expresivă și de reprezentare a limbajului de implementare
- previne explozia în dimensiune a expresiilor
- metodă exactă (limitată doar de precizia de reprezentare)

Derivarea a funcțiilor compuse

Dacă f este o funcție compusă:

$$g : \mathbb{R}^k \to \mathbb{R}^m$$

$$h : \mathbb{R}^n \to \mathbb{R}^k$$

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

$$f = g \circ h = g(h(x))$$

atunci conform regulii de derivare a funcțiilor compuse:

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

Metode de calcul – forward mode

- parcurge "lanțul" de derivate de la stânga la dreapta (intrare-ieșire)
- ordinea de evaluare a expresiei corespunde cu ordinea de evaluare a derivatelor

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

- valorile sunt calculate în tandem fără a necesita memorie suplimentară
- în fiecare iterație intrare-ieșire:

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

- implementare uzuală cu *numere duale* Clifford [8] (denumite și jets)

Metode de calcul – reverse mode

- parcurge “lanțul” de derivate de la dreapta la stânga (ieșire-intrare)
- necesită memorie suplimentară pentru valorile intermediare (“tape”)
- derivata variabilei dependente în funcție de variabila intermediată

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

- algoritm în două etape
  1. parcurgere înainte: evaluare și populare valori intermediare
  2. parcurgere înapoi: propagarea valorilor de tip “adjoint”
- conform regulii de derivare a funcțiilor compuse:

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

Complexitate algoritmică

A. Griewank and A. Walther [9] (cap 4, secțiuni 4.5 și 4.6)

$$\text{ops}(\text{forward}) \approx c_1 \cdot \text{ops}(f), c_1 \in [2, 2.5]$$

$$\text{ops}(\text{reverse} \approx c_2 \cdot \text{ops}(f), c_2 \in [3, 4]$$

Numărul de operații necesare pentru a calcula o matrice Jacobiană de dimensiuni $m \times n$

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

Învățare automată

- necesită calculul unui ∇ al unei funcții obiectiv scalare cu un număr mare de parametri
- din acest motiv modul reverse reprezintă abordarea cea mai eficientă

Graf computațional

Reprezentarea calculelor sub formă de graf F. L. Bauer [10] stă la baza diferențierii automate.

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

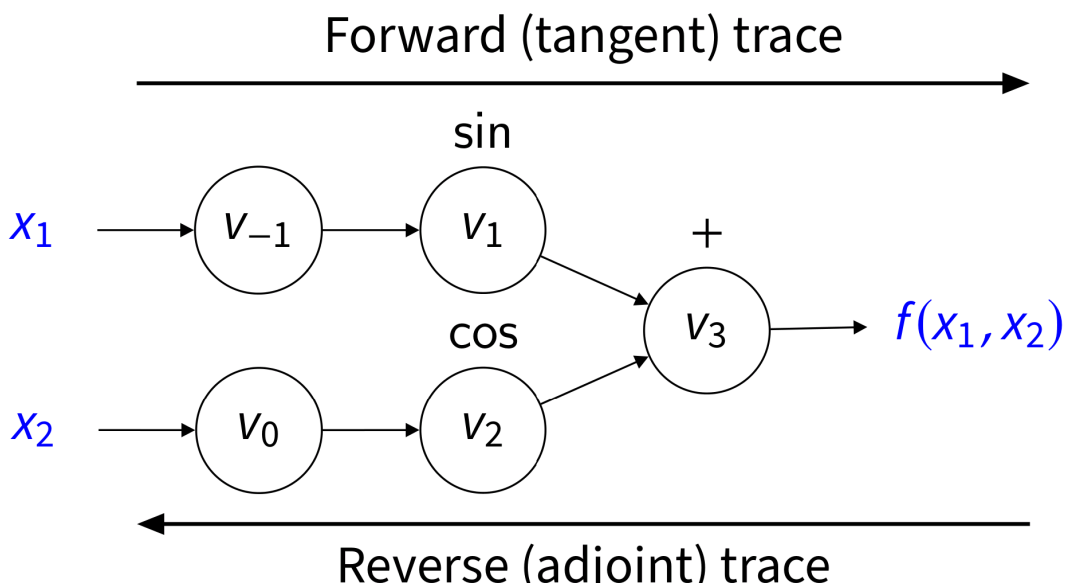


Figure 4: Notație cf. A. Walther [11]

Graf computațional

Presupunem că vrem să calculăm valoarea gradientului pentru $(x_1, x_2) = (2, 3)$

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

```py
import jax
import jax.numpy as jnp
def f1(x):
  return jnp.sin(x[0]) + jnp.cos(x[1])

grad = jax.value_and_grad(f1) grad([2.0, 3.0])
```

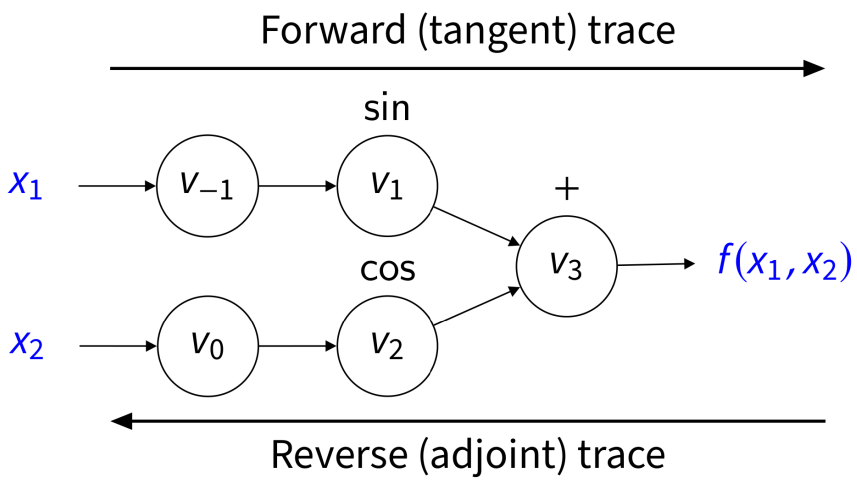
```
(Array(-0.08069509, dtype=float32, weak_type=True),
[Array(-0.41614684, dtype=float32, weak_type=True),
Array(-0.14112, dtype=float32, weak_type=True)])
```

Exemplu 1: calcul forward mode

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

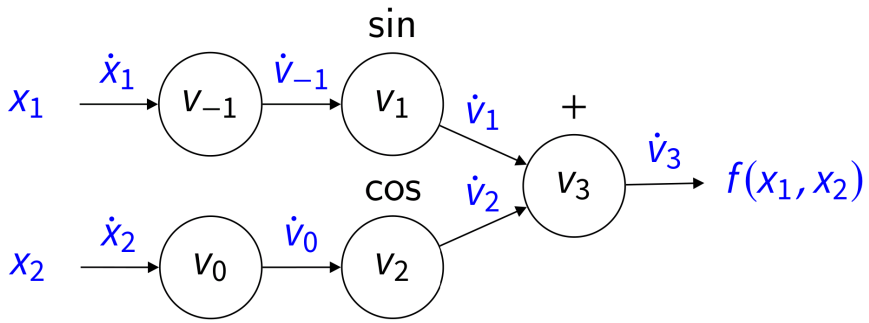
Evaluarea grafului

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



Exemplu 1: calcul forward mode

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



$$\dot{v}_{-1} = \dot{x}_1 = 1$$

$$\dot{v}_0 = \dot{x}_2 = 0$$

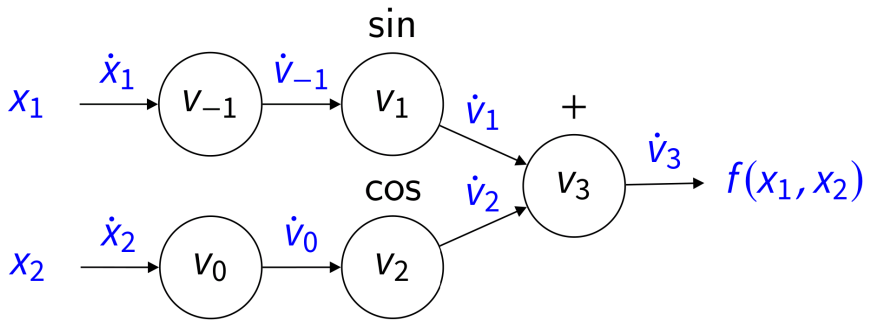
$$\dot{v}_1 = \dot{v}_{-1} \cdot \cos(v_{-1}) = 1 \cdot \cos(2) = -0.416$$

$$\dot{v}_2 = \dot{v}_0 \cdot -\sin(v_0) = 0 \cdot -\sin(3) = 0$$

$$\dot{v}_3 = \dot{v}_1 + \dot{v}_2 = -0.416 + 0 = -0.416$$

Exemplu 1: calcul forward mode

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



$$\dot{v}_{-1} = \dot{x}_1 = 0$$

$$\dot{v}_0 = \dot{x}_2 = 1$$

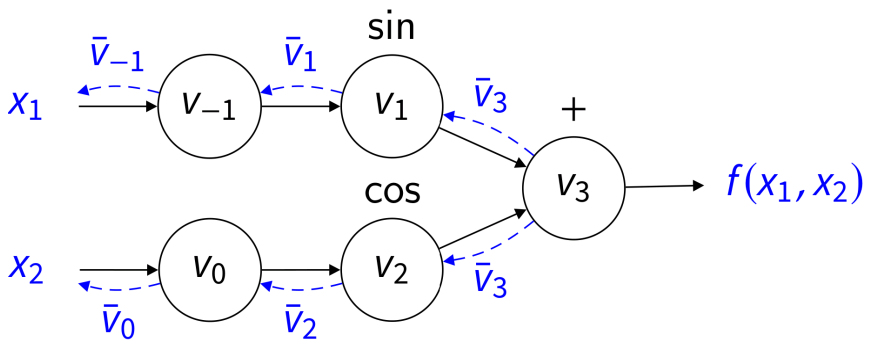
$$\dot{v}_1 = \dot{v}_{-1} \cdot \cos(v_{-1}) = 0 \cdot \cos(2) = 0$$

$$\dot{v}_2 = \dot{v}_0 \cdot -\sin(v_0) = 1 \cdot -\sin(3) = -0.141$$

$$\dot{v}_3 = \dot{v}_1 + \dot{v}_2 = 0 - 0.141 = -0.141$$

Exemplu 1: calcul reverse mode

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



$$\bar{v}_3 = 1$$

$$\bar{v}_2 = \bar{v}_3 \cdot \partial v_3 / \partial v_2 = 1 \cdot 1 = 1$$

$$\bar{v}_1 = \bar{v}_3 \cdot \partial v_3 / \partial v_1 = 1 \cdot 1 = 1$$

$$\bar{v}_0 = \bar{v}_2 \partial v_2 / \partial v_0 = 1 \cdot -\sin(3) = -0.141$$

$$\bar{v}_{-1} = \partial v_1 / \partial v_{-1} = 1 \cdot \cos(2) = -0.416$$

Exemplu 2:

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

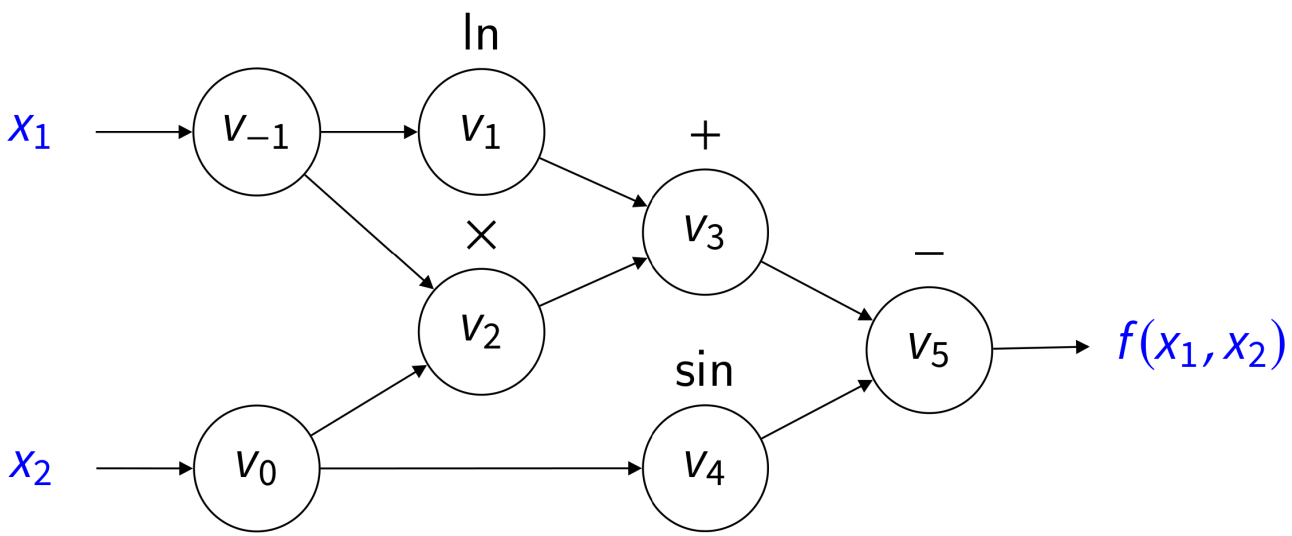


Figure 9: A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind [6]

Exemplu 2

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

Evaluarea grafului computațional la $(x_1, x_2) = (2, 5)$

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

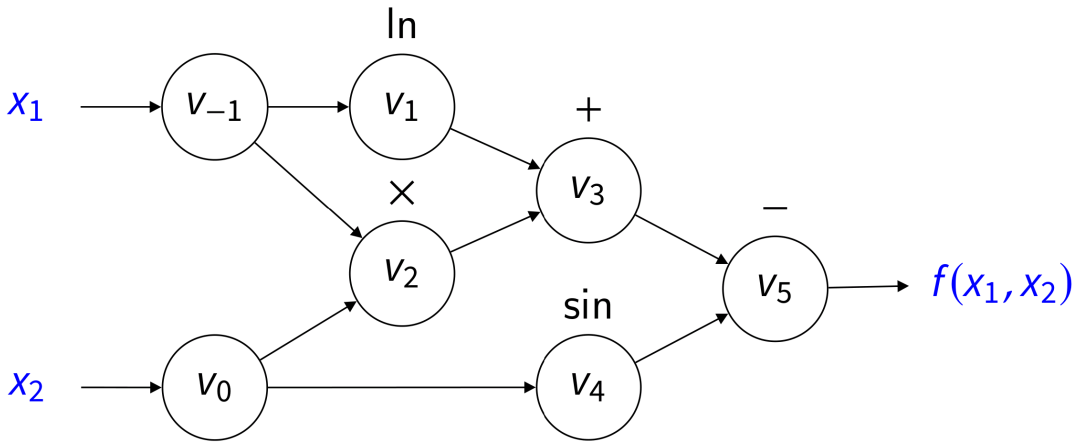


Figure 10: A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind [6]

Exemplu 2

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

Parcurs înainte pentru $\dot{x}_1$, $(x_1, x_2) = (2, 5)$

$$\dot{v}_{-1} = \dot{x}_1 = 1$$

$$\dot{v}_0 = \dot{x}_2 = 0$$

$$\dot{v}_1 = \dot{v}_{-1}/v_{-1} = 0.5$$

$$\dot{v}_2 = \dot{v}_{-1} v_0 + \dot{v}_1 v_{-1} = 1 \cdot 5 + 0 \cdot 2 = 5 \dot{v}_3 =$$

$$\dot{v}_4 = \dot{v}_0 \cdot \cos(v_4) = 0 \cdot \cos(5) = 0$$

$$\dot{v}_5 = \dot{v}_3 - \dot{v}_4 = 5.5 - 0 = 5.5$$

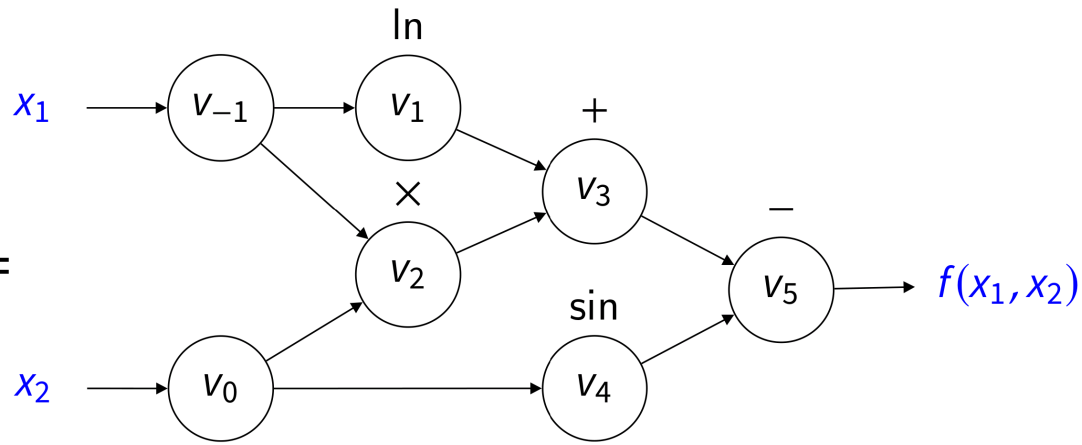


Figure 11: A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind [6]

Exemplu 2

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

Parcurs înapoi pentru $\dot{x}_1$, $(x_1, x_2) = (2, 5)$

$$\overline{v}_5 = \overline{f} = 1$$

$$\overline{v}_4 = \overline{v}_5 \partial v_5 / \partial v_4 = 1 \cdot -1 = -1$$

$$\overline{v}_3 = \overline{v}_5 \partial v_5 / \partial v_3 = 1 \cdot 1 = 1$$

$$\overline{v}_2 = \overline{v}_3 \partial v_3 / \partial v_2 = 1 \cdot 1 = 1$$

$$\overline{v}_1 = \overline{v}_3 \partial v_3 / \partial v_1 = 1 \cdot 1 = 1$$

$$\overline{v}_0 = \overline{v}_2 \partial v_2 / \partial v_0 + \overline{v}_4 \partial v_4 / \partial v_0 = 1.716$$

$$\overline{v}_{-1} = \overline{v}_2 \partial v_2 / \partial v_{-1} + \overline{v}_1 \partial v_1 / \partial v_{-1} = 5.5$$

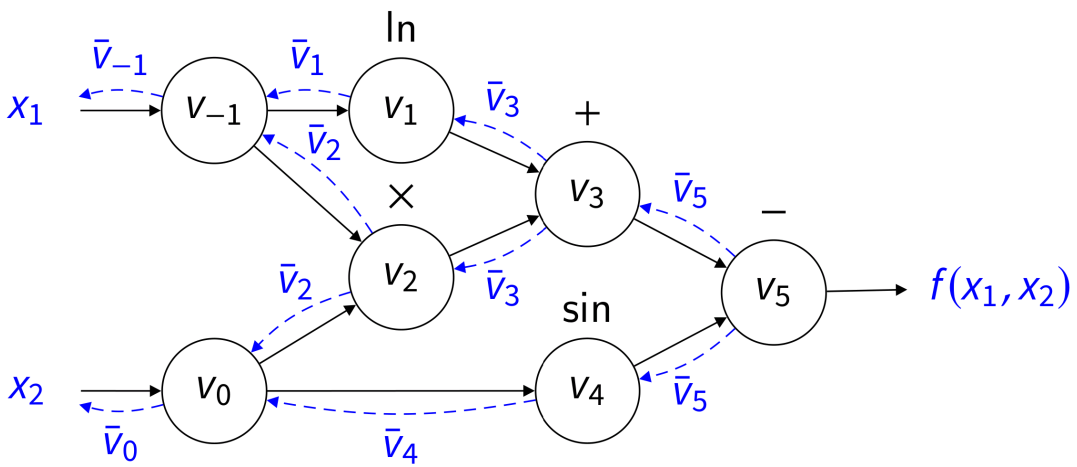


Figure 12: A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind [6]

Optimizarea coeficienților funcțiilor

$f(\theta) = \theta_1 \cos(\theta_2) + \theta_3 \sin(\theta_4)$, calculăm gradientul în punctul $\theta = (0.5, 2, 0.7, 3)$

$v_{-3} = \theta_1 = 0.5$

$v_{-2} = \theta_2 = 2.0$

$v_{-1} = \theta_3 = 0.7$

$v_0 = \theta_4 = 3.0$

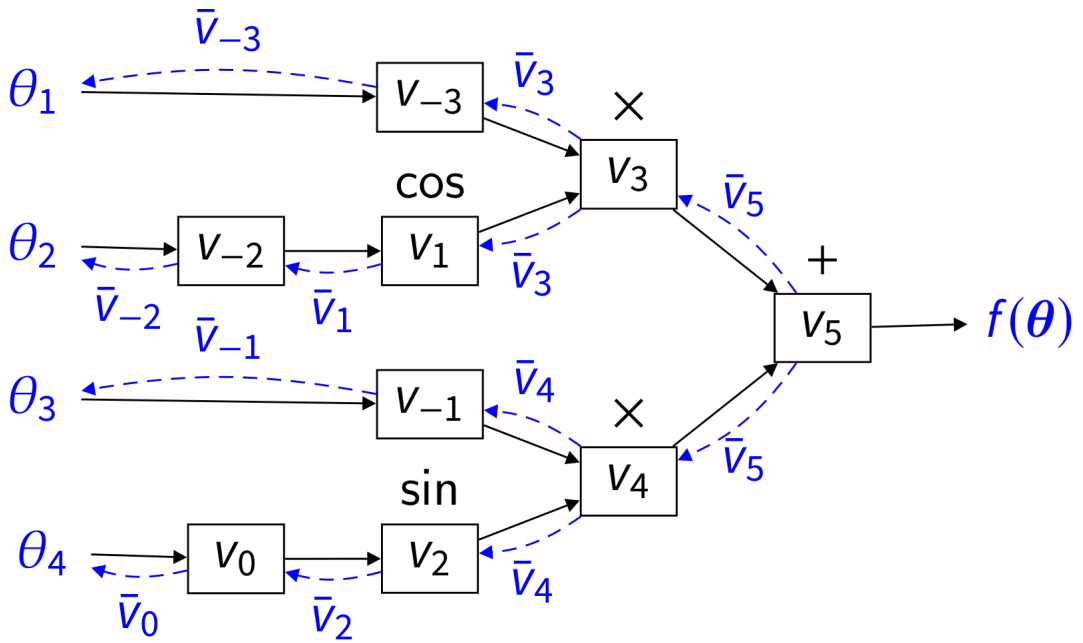
$v_1 = \cos(v_{-2}) = -0.416$

$v_2 = \sin(v_0) = 0.141$

$v_3 = v_{-3} \cdot v_1 = -0.208$

$v_4 = v_{-1} \cdot v_2 = 0.099$

$v_5 = v_3 + v_4 = -0.109$



Optimizarea coeficienților funcțiilor

$f(\theta) = \theta_1 \cos(\theta_2) + \theta_3 \sin(\theta_4)$, calculăm gradientul în punctul $\theta = (0.5, 2, 0.7, 3)$

$$\overline{v}_5 = \overline{f} = 1$$

$$\overline{v}_4 = \overline{v}_5 \partial v_5 / \partial v_4 = 1 \cdot 1 = 1$$

$$\overline{v}_3 = \overline{v}_5 \partial v_5 / \partial v_3 = 1 \cdot 1 = 1$$

$$\overline{v}_2 = \overline{v}_4 \partial v_4 / \partial v_2 = 1 \cdot v_{-1} = 0.7$$

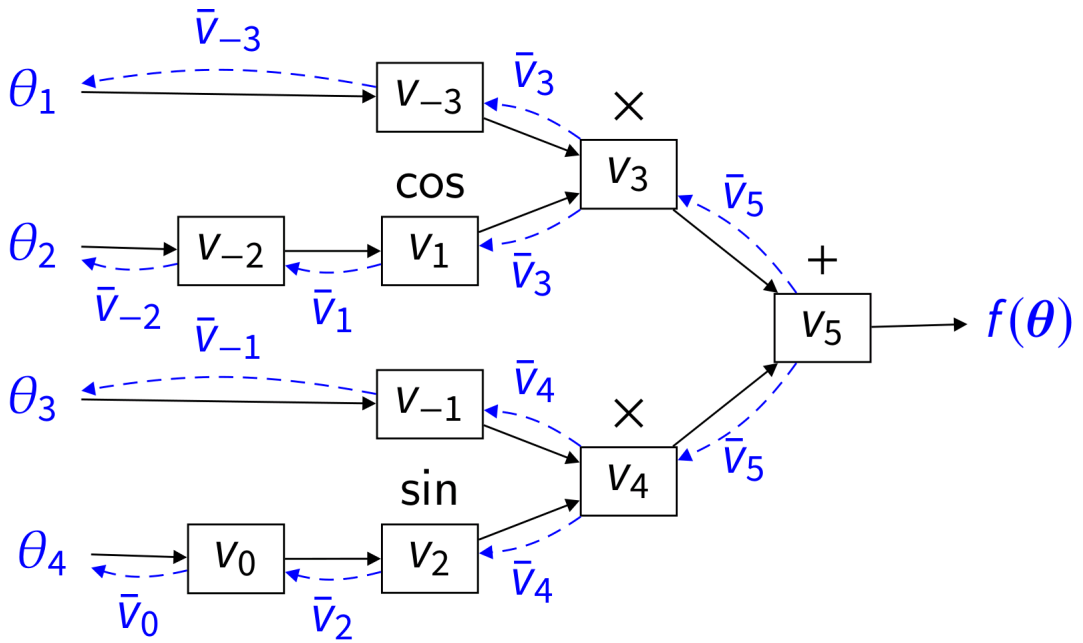
$$\overline{v}_1 = \overline{v}_3 \partial v_3 / \partial v_1 = 1 \cdot v_{-3} = 0.5$$

$$\overline{v}_0 = \overline{v}_2 \partial v_2 / \partial v_0 = 0.7 \cdot \cos(v_0) = -0.693$$

$$\overline{v}_{-1} = \overline{v}_4 \partial v_4 / \partial v_{-1} = 1 \cdot v_2 = 0.141$$

$$\overline{v}_{-2} = \overline{v}_1 \partial v_1 / \partial v_{-2} = 0.5 \cdot -\sin(v_{-2}) = -0.454$$

$$\overline{v}_{-3} = \overline{v}_3 \partial v_3 / \partial v_{-3} = 1 \cdot v_1 = -0.416$$



Optimizarea coeficienților funcțiilor

Îmbunătățirea consumului de memorie

Introducerea coeficienților multiplicativi în interiorul nodurilor.

- unele valori intermediare nu vor mai fi prezente în graf
- aceste valori vor trebui recalculate (prin împărțire la coeficientul θ corespunzător)

avantaj consum redus de memorie, mai puține noduri de vizitat
dezavantaj recalcularea valorilor intermediare

Checkpointing

Recalcularea valorilor între checkpoints (în loc să fie stocate în memorie)

Optimizarea coeficienților funcțiilor

Îmbunătățirea consumului de memorie

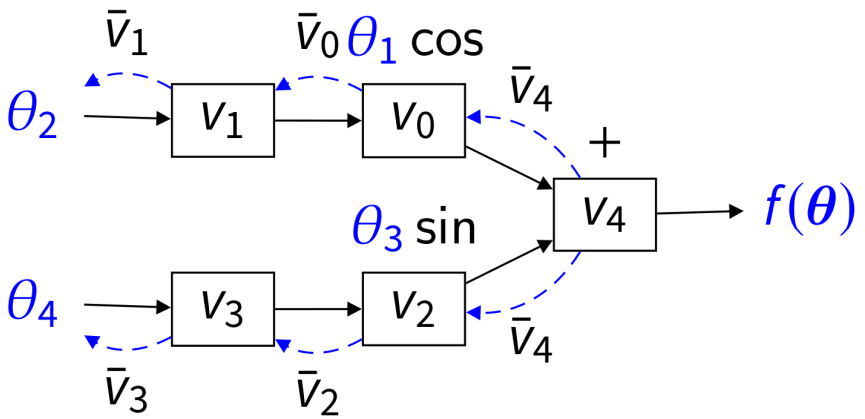
$$\bar{v}_4 = \bar{f} = 1$$

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

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

$$\bar{v}_3 = \bar{v}_2 \frac{\partial v_2}{\partial v_3} = \theta_3 \cdot \frac{\theta_1 \cos(3)}{\theta_1} = -0.693$$

$$\bar{v}_1 = \bar{v}_0 \frac{\partial v_0}{\partial v_1} = \theta_1 \cdot -\frac{\theta_3 \sin(2)}{\theta_3} = -0.454$$



Paradigme de implementare

- Funcții elementare (înlocuirea operațiilor matematice cu apeluri de funcții de bibliotecă)
- Supraîncărcarea operatorilor
  - 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
  - XAD https://auto-differentiation.github.io/
- Bazat pe compilator
  - 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

Multe abordări și optimizări posibile C. C. Margossian [12].

Exemplu de implementare – numere duale (C++)

Număr dual

$$d = a + b\varepsilon, a, b \in \mathbb{R}, \varepsilon^2 = 0, \varepsilon \neq 0$$

```cpp
struct dual {
  double a{0};
  double b{0};
  auto operator*() const -> std::tuple<double, double> { return {a, b}; }

  // supraîncărcarea operatorilor pentru toate operațiile de bază
}
```

Adunare

```cpp
friend auto operator+(dual const& lhs, dual const& rhs) -> dual {
  auto const [a, b] = *lhs;
  auto const [c, d] = *rhs;
  return {a+c, b+d};
}

friend auto operator+(std::floating_point auto x, dual const& a) {
  return dual{x} + a;
}

friend auto operator+(dual const& a, std::floating_point auto x) {
  return a + dual{x};
}
```

Scădere

```cpp
friend auto operator-(dual const& lhs, dual const& rhs) -> dual {
  auto const [a, b] = *lhs;
  auto const [c, d] = *rhs;
  return {a-c, b-d};
}

friend auto operator-(std::floating_point auto x, dual const& a) {
  return dual{x} - a;
}

friend auto operator-(dual const& a, std::floating_point auto x) {
  return a - dual{x};
}
```

Înmulțire

```cpp
friend auto operator*(dual const& lhs, dual const& rhs) -> dual {
  auto const [a, b] = *lhs;
  auto const [c, d] = *rhs;
  return {a*c, a*d+b*c};
}
friend auto operator*(std::floating_point auto x, dual const& a) {
  return dual{x} * a;
}
friend auto operator*(dual const& a, std::floating_point auto x) {
  return a * dual{x};
}
```

Împărțire

```cpp
friend auto operator/(dual const& lhs, dual const& rhs) -> dual {
  auto const [a, b] = *lhs;
  auto const [c, d] = *rhs;
  return {a/c, (b*c-a*d)/(c*c)};
}

friend auto operator/(std::floating_point auto x, dual const& a) {
  return dual{x} / a;
}

friend auto operator/(dual const& a, std::floating_point auto x) {
  return a / dual{x};
}
```

Alte operații

```cpp
auto sin() const -> dual {
  auto const [a, b] = this->operator*();
  return {std::sin(a), b * std::cos(a)};
}
auto cos() const -> dual {
  auto const [a, b] = this->operator*();
  return {std::cos(a), -b * std::sin(a)};
}

auto exp() const -> dual {
  auto const [a, b] = this->operator*();
  return {std::exp(a), b * std::exp(a)};
}
```

Utilizare

$f(x, y) = x \cdot y + \sin(x), (x_1, x_2) = (0.5, 4.2)$

```cpp
std::array duals { dual{0.5}, dual{4.2} };

std::array gradient {0.0, 0.0};
for (auto i = 0; i < duals.size(); ++i) {
  duals[i].b = 1;
  auto c = duals[0] * duals[1] + duals[0].sin();
  gradient[i] = c.b;
  duals[i].b = 0;
}
std::cout << "\nFORWARD MODE:\n";
std::cout << "df/dx: " << gradient[0] << "\n"; // df/dx = 5.07758
std::cout << "df/dy: " << gradient[1] << "\n"; // df/dy = 0.5
```

Exemplu de implementare – memorie/“tape” (C++)

Difefențiere automată în mod reverse folosind o casetă de memorie (“tape”).

Ingrediente necesare:

- casetă de memorie (“tape”)
- posibilitatea de a reprezenta un graf (noduri și muchii)
- clasă reprezentând o variabilă cu supraîncărcarea operatorilor
  (pentru simplitate, se vor considera doar operații nulare, unare, binare)

La fiecare operație, nodul produs se va salva în memorie (“tape”).

Gradientul se obține traversând caseta de memorie în sens invers.

Structura unui nod în caseta de memorie

```cpp
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
};

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

Caseta de memorie

```cpp
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 length() const -> std::size_t { return std::size(nodes); }
    auto variable(T value) { return Var<T> {*this, value, push()}; }
    auto clear() { nodes.clear(); }
};
```

Variabilă diferențiabilă

```cpp
template<Arithmetic T>
struct Var {
    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];
            auto d = grad[i];

            for (auto j = 0UL; j < std::size(n.inputs); ++j)
                grad[n.inputs[j]] += n.partials[j] * d;
        }
        return {grad};
    }
    // + operatori supraîncărcați pentru toate operațiile uzuale
}
```

```cpp
template<Arithmetic T>
struct Var { // continuare (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 { // continuare (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)};
    }
}
```

Utilizare

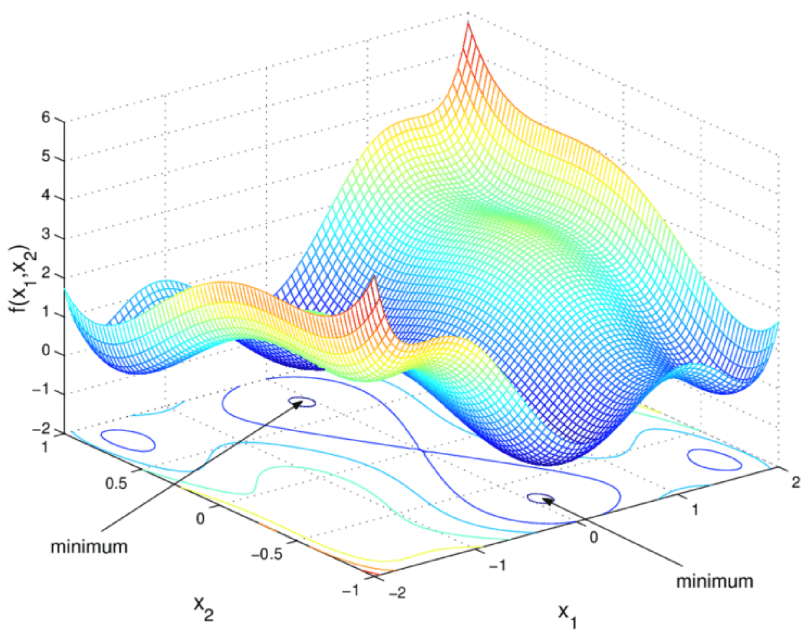
Fie functia $f(x, y) = \log(x) + xy - \sin(y)$, în punctul $(x, y) = (2, 5)$

```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));        // df/dx = 5.5
  expect(eq(g.wrt(y), a - std::cos(b)));  // df/dy = 1.716
};
```

Optimizare – netoda neliniară a celor mai mici pătrate

Fie $\boldsymbol{x} \in \mathbb{R}^n$ și $F(\boldsymbol{x}) = \left[f_1(x), \ldots, f_{m(x)}\right]^\top \in \mathbb{R}^m$. Dorim să aflăm minimum funcției:

$$\operatorname{argmin}_x \frac{1}{2} \|F(\boldsymbol{x})\|^2$$



Exemplu: NIST – Thurber dataset

Semiconductor electron mobility modeling

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

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

- o variabilă de ieșire y
- o variabilă de intrare x
- sapte parametri $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_7)$ to be estimated
- 37 observații $(x_i, y_i), i = 1, \ldots, 37$
- erori heteroscedastice ϵ
- nivel ridicat de dificultate datorită naturii neliniare a modelului

Semiconductor electron mobility modeling

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

Implementare (1)

```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(); }
  // to be continued
};
```

Implementare (2)

```cpp
struct thurber_functor {
  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());
  }
  auto operator()(auto const& input, auto* residual, auto* jacobian) const ->
  int { /* calculul gradientului se va face aici*/ }
};
```

Implementare (3)

```cpp
auto operator()(auto const& input, auto* residual, auto* jacobian) const ->
int {
  for (auto i = 0; i < std::ssize(xval); ++i) {
    reverse::Tape<Scalar> tape;
    std::vector<decltype(tape)::Variable> beta;
    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);
    residual[i] = f.value - yval[i];
    auto g = f.gradient();
    for (auto& 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));
}
```

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

- Convergență în 20 de iterații, cu un număr maxim de 50 de evaluări ale funcției de cost
- Rezultatul este în concordanță cu cel oferit de NIST
- Gradientul este calculat eficient folosind autodiff în mod reverse (implementare proprie)

Performanță și scalabilitate

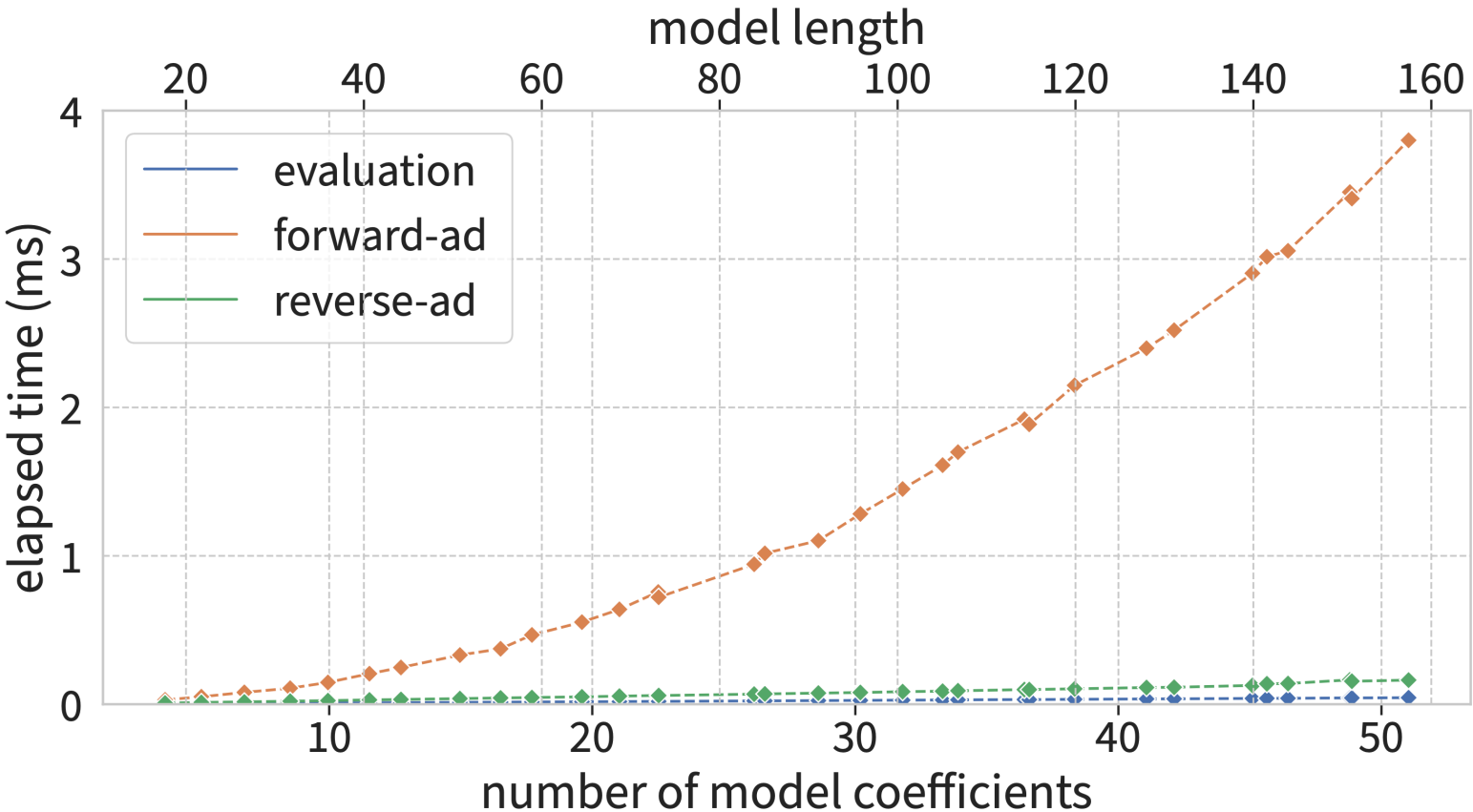


Figure 17: Performanță evaluare funcție și gradient, CPU: 5950X

Concluzii

Programarea folosind derivate manuale este laborioasă și predispusă la erori.

Diferentierea automată este un instrument puternic pentru optimizarea funcțiilor neliniare.

Folosind diferentierea automată, putem calcula derivate exacte ale funcțiilor.

- mod forward: eficient pentru funcții cu mai multe ieșiri
- mod reverse: eficient pentru funcții cu mai multe intrări
- putem deriva programe (bucle, expresii condiționale)

Principiul gradientului ieftin si implicațiile sale în optimizare nu sunt suficient cunoscute.

Cod sursă

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

Operon autodiff module

https://github.com/heal-research/operon/tree/main/include/operon/interpreter

Differential programming @ NeurIPS 2021

https://diffprogramming.mit.edu/

Autodiff workshop @ NeurIPS 2016

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

Bibliography

[1] R. E. Wengert, “A Simple Automatic Derivative Evaluation Program,” *Commun. ACM*, vol. 7, no. 8, pp. 463–464, Aug. 1964, doi: 10.1145/355586.364791.

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

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986.

[4] P. J. Werbos, “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences,” Doctoral dissertation, 1974.

[5] Hecht-Nielsen, “Theory of the backpropagation neural network,” in *International 1989 Joint Conference on Neural Networks*, 1989, pp. 593–605vol.1. doi: 10.1109/IJCNN.1989.118638.

[6] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic Differentiation in Machine Learning: A Survey,” *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 5595–5637, Jan. 2017.

[7] F. Wang, D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf, “Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator,” *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, July 2019, doi: 10.1145/3341700.

[8] Clifford, “Preliminary Sketch of Biquaternions,” *Proceedings of The London Mathematical Society*, pp. 381–395, 1871.

[9] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Second. USA: Society for Industrial, Applied Mathematics, 2008.

[10] F. L. Bauer, “Computational Graphs and Rounding Error,” *SIAM Journal on Numerical Analysis*, vol. 11, no. 1, pp. 87–96, 1974, doi: 10.1137/0711010.

[11] A. Walther, “Getting Started with ADOL-C,” in *Combinatorial Scientific Computing*, 2009.

[12] C. C. Margossian, “A review of automatic differentiation and its efficient implementation,” *WIREs Data Mining and Knowledge Discovery*, vol. 9, no. 4, Mar. 2019, doi: 10.1002/widm.1305.