Automatic Differentiation

A tale of two languages

Guillaume Dalle

2024-12-03

Introduction

Slides

https://gdalle.github.io/PyData2024-AutoDiff/

Motivation

What is a derivative?

A linear approximation of a function around a point.

Why do we care?

Derivatives of complex programs are essential in optimization and machine learning.

What do we need to do?

Not much: Automatic Differentiation (AD) computes derivatives for us!

Bibliography

- Blondel and Roulet (2024): the most recent book
- Griewank and Walther (2008): the bible of the field
- Baydin et al. (2018), Margossian (2019): concise surveys

Understanding AD

Derivatives: formal definition

Derivative of f at point x: linear map $\partial f(x)$ such that

$$f(x+v) = f(x) + \partial f(x)[v] + o(\|v\|)$$

In other words,

$$\partial f(x)[v] = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon v) - f(x)}{\varepsilon}$$

Various flavors of differentiation

- **Manual**: work out ∂f by hand
- **Numeric**: $\partial f(x)[v] \approx \frac{f(x+\varepsilon v)-f(x)}{\varepsilon}$
- **Symbolic**: enter a formula for f, get a formula for ∂f
- **Automatic**[1]: code a program for f, get a program for $\partial f(x)$

[1] or algorithmic

Two ingredients of AD

Any derivative can be obtained from:

1. Derivatives of **basic functions**: $\exp, \log, \sin, \cos, \ldots$
2. Composition with the **chain rule**:

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

or its adjoint[2]

$$\partial(f \circ g)^*(x) = \partial g(x)^* \circ \partial f(g(x))^*$$

[2] the "transpose" of a linear map

What about Jacobian matrices?

We could multiply matrices instead of composing linear maps:

$$J_{f \circ g}(x) = J_f(g(x)) \cdot J_g(x)$$

where the Jacobian matrix is

$$J_f(x) = \left(\partial f_i / \partial x_j\right)_{i,j}$$

- very wasteful in high dimension (think of $f = \mathrm{id}$)
- ill-suited to arbitrary spaces

Matrix-vector products

We don't need Jacobian matrices as long as we can compute their products with vectors:

Jacobian-vector products

$$J_f(x)v = \partial f(x)[v]$$

Propagate a perturbation v from input to output

Vector-Jacobian products

$$w^\top J_f(x) = \partial f(x)^*[w]$$

Backpropagate a sensitivity w from output to input

Forward mode

Consider $f = f_L \circ \ldots \circ f_1$ and its Jacobian $J = J_L \cdots J_1$.

Jacobian-vector products decompose from layer 1 to layer L:

$$J_L(J_{L-1}(\ldots \underbrace{J_2(\underbrace{J_1 v}_{v_1})}_{v_2}))$$

Forward mode AD relies on the chain rule.

Reverse mode

Consider $f = f_L \circ \ldots \circ f_1$ and its Jacobian $J = J_L \cdots J_1$.

Vector-Jacobian products decompose from layer L to layer 1:

$$\underbrace{((\underbrace{(w^\top J_L}_{w_L})J_{L-1}}_{w_{L-1}} \ldots)J_2)J_1$$

Reverse mode AD relies on the adjoint chain rule.

Jacobian matrices are back

Consider $f : \mathbb{R}^n \to \mathbb{R}^m$. How to recover the full Jacobian?

Forward mode

Column by column:

$$J = \begin{pmatrix} Je_1 & \dots & Je_n \end{pmatrix}$$

where e_i is a basis vector.

Reverse mode

Row by row:

$$J = \begin{pmatrix} e_1^\top J \\ \vdots \\ e_m^\top J \end{pmatrix}$$

Complexities

Consider $f : \mathbb{R}^n \to \mathbb{R}^m$. How much does a Jacobian cost?

Theorem

Each JVP or VJP takes as much time and space as $O(1)$ calls to f.

sizes	jacobian	forward	reverse	best mode
generic	jacobian	$O(n)$	$O(m)$	depends
$n = 1$	derivative	$O(1)$	$O(m)$	forward
$m = 1$	gradient	$O(n)$	$O(1)$	reverse

Fast reverse mode gradients make deep learning possible.

Using AD

Three types of AD users

1. **Package users** want to differentiate through functions
2. **Package developers** want to write differentiable functions
3. **Backend developers** want to create new AD systems

Python vs. Julia: users

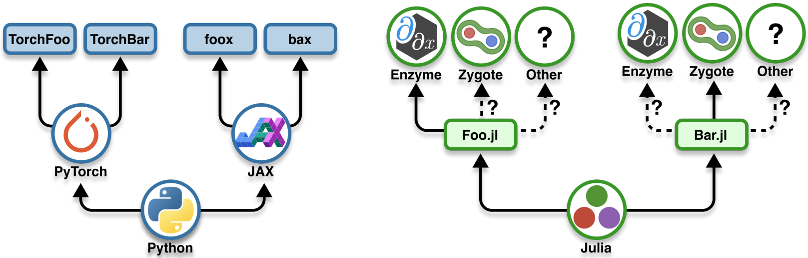

Image: courtesy of Adrian Hill

Python vs. Julia: developers

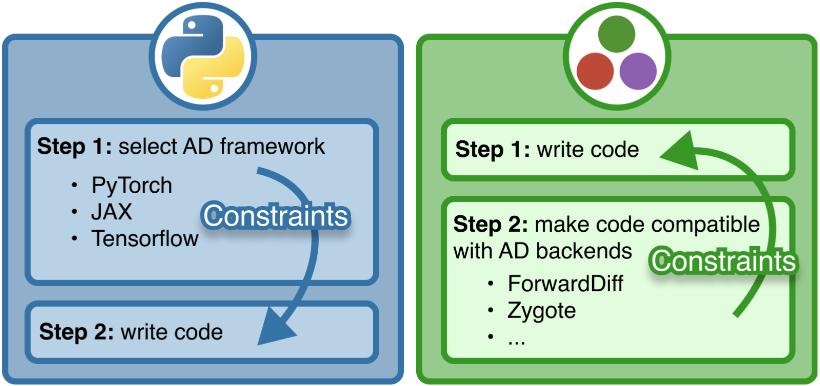

Image: courtesy of Adrian Hill

Why so many packages?

- Conflicting **paradigms**:
  - numeric vs. symbolic vs. algorithmic
  - operator overloading vs. source-to-source
- Cover varying **subsets of the language**
- Historical reasons: developed by **different people**

Full list available at https://juliadiff.org/.

Conclusion

Going further

- ☒ AD through a simple function
- ☐ AD through an expectation (Mohamed et al. 2020)
- ☐ AD through a convex solver (Blondel et al. 2022)
- ☐ AD through a combinatorial solver (Mandi et al. 2024)

Take-home message

Computing derivatives is **automatic** and efficient.

Each AD system comes with **limitations**.

Learn to recognize and overcome them.

References

Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. "Automatic Differentiation in Machine Learning: A Survey." *Journal of Machine Learning Research* 18 (153): 1–43. http://jmlr.org/papers/v18/17-468.html.

Blondel, Mathieu, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. 2022. "Efficient and Modular Implicit Differentiation." In *Advances in Neural Information Processing Systems*. https://openreview.net/forum?id=Q-HOv_zn6G.

Blondel, Mathieu, and Vincent Roulet. 2024. "The Elements of Differentiable Programming." arXiv. https://doi.org/10.48550/arXiv.2403.14606.

Griewank, Andreas, and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics.