Automatic Differentiation for Scientists

How I learned to stop worrying and love the chain rule

Kiran Shila

July 22, 2024

Who is this person

- 4th year PhD student in EE
- From Tampa, FL (USF grad)
- Radio astronomy
- Open-source software fan
- Rust, Julia, Lisp

Why we want derivatives

Derivatives are important in many areas of science
(why else would they teach us Calculus?)

- Optimization and parameter estimation
  - Gradient descent and all of its forms require a gradient
  - Encompasses all of machine learning, nonlinear programming, etc.
  - HMC needs gradients (the Hamiltonian part)
- Uncertainty quantification and sensitivity analysis
  - Derivatives let us see how changes in the input propagate
  - Used in controls, ecology, pharmacology
- Inverse problems
  - Couples these concepts to solve novel designs
  - Medical imaging, black hole imaging, photonics, geophysics

How do we compute derivatives

- The definition

$$f(x) = \lim_{h \to \infty} \frac{A(x+h) - A(x)}{h} := A'(x) \tag{1}$$

- Derived rules
  - Power rule
  - Quotient rule
  - Etc.
- Lookup tables
  - $\sin \to \cos$
  - $\log_a(x) \to \frac{1}{x \ln(a)}$
  - Etc.

How do we compute derivatives

Ok so,

- We get a nice elegant symbolic expression
- Useful for some stuff
- Many other cases (gradient descent), it doesn't matter. What we want is the **value**

How to teach a computer calculus

Say you're handed a random program and you want its derivative.
What do you do?

$$f(x) = \lim_{h \to \infty} \frac{A(x+h) - A(x)}{h} := A'(x) \tag{2}$$

And put on our engineer's hat

$$A'(x) \approx \frac{A(x+h) - A(x)}{h} \tag{3}$$

Finite Differencing Bad

- This is not good and you should avoid it
- Numbers in computers have finite precision
- FD breaks the cardinal sins of numerical code
    - Don't add small numbers and big numbers
    - Don't divide small numbers and big numbers

Finite Differencing Bad

Python example:

```
>>> f = lambda x: x**2
>>> h = sys.float_info.epsilon
```

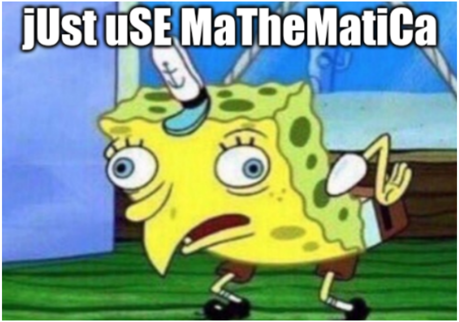
What does this return

```
>>> (f(2+h) - f(2)) / h
0.0
```

Ok what about

```
>>> h = 1
>>> (f(2+h) - f(2)) / h
5.0
```

Computer Algebra Systems



- Symbolic solutions can be slow
- We probably don't need a symbolic results
- Do we really need to give Stephen more money?

Enter: Automatic Differentiation

- For decades, applied math researchers have written about "Automatic (Algorithmic) Differentiation" but no one noticed.
- High-level description: Nonstandard evaluation of a computer program that computes exact derivates alongside normal evaluation
- Machine learning people were designing models that they could solve gradients of **by hand** until the mid-2010s
- Now with all the money involved, AD has picked up and is a essential tool in ML (PyTorch (2017))
- This tool isn't just for ML: Computing the derivatives of arbitrary programs will become an essential tool in science

Ok, but what is it

- Literally, just the chain rule because computer programs **are** function composition
  Recall...

$$\frac{d}{dx}\left[f(g(x))\right] = f'\left(g\left(x\right)\right)g'\left(x\right) \tag{4}$$

- Let's translate this to Python

```python
def g(x):
    ...

def f(x):
    ...

def my_fun(x):
    a = g(x)
    b = f(a)
    return b
```

We love graphs

$$\begin{tikzcd} \boxed{\mathrm{x}} \arrow[d, "p_1 = x"'] \\ \mathrm{g} \arrow[d, "p_2 = g(p_1)"'] \\ \mathrm{f} \arrow[d, "p_3 = f(p_2)"'] \\ \boxed{\mathrm{f(g(x))}} \end{tikzcd}$$

We love augmented graphs

Reminder

$$\frac{d}{dx}\left[f(g(x))\right] = f'(g(x))\, g'(x)$$

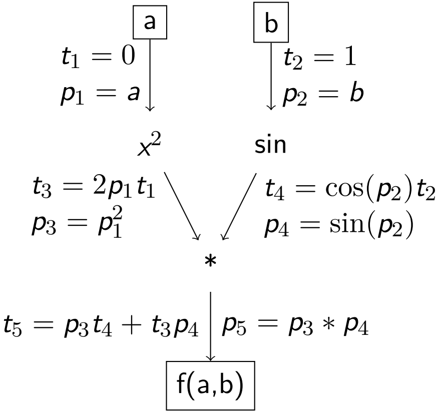
$$\begin{tikzcd}
\boxed{\mathrm{x}} \arrow[d, "p_1 = x"', "t_1 = 1"] \\
\mathrm{g} \arrow[d, "p_2 = g(p_1)"', "t_2 = g'(p_1)t_1"] \\
\mathrm{f} \arrow[d, "p_3 = f(p_2)"', "t_3 = f'(p_2)t_2"] \\
\boxed{\mathrm{f(g(x))}}
\end{tikzcd}$$

Concrete Example


```
def f(a,b):
    x = a**2
    y = sin(b)
    return x * y
```

Concrete Example

```python
def f(a,b):
    x = a**2
    y = sin(b)
    return x * y
```



Yes but how

- We formed this graph view of the problem, but we didn't need to
- We just need a data structure to hold the value and the tangent
- Then we define the rules for math/calculus of this new data structure

Dual Numbers

- The most popular approach is the "dual number"

$$a + b\epsilon$$

where

$$\epsilon^2 = 0$$

- Lots of cool math implications you should look up on "hypercomplex" numbers

Python Dual

```python
class Dual:
    def __init__(self, primal,tangent):
        self.primal = primal
        self.tangent = tangent
    def __add__(self,other):
        primal = self.primal + other.primal
        tangent = self.tangent + other.tangent
        return Dual(primal, tangent)
    def __mul__(self, other):
        primal = self.primal * other.primal
        tangent = self.primal  * other.tangent + \
                  self.tangent * other.primal
        return Dual(primal, tangent)
    def __repr__(self):
        return f"{self.primal} + {self.tangent}ϵ"
```

Python Dual

Let's define $f(x) = 3x^2$

```
def f(x):
    y = x * x
    z = y + y + y
    return z
>>> f(3)
27
>>> f(Dual(3,1))
27 + 18ϵ
>>> Dual(0,1) * Dual(0,1)
0 + 0ϵ
```

Important Points - Forward Mode AD

- Obviously incomplete implementation, but simple (no need to actually build a graph)
- Missing promotion between non-Dual types
- What about $\sin$, $\log$, etc.
- O(n) complexity (same as finite difference) for $f : \mathbb{R}^n \to \mathbb{R}^m$

Another formulation

- Let's look again at the chain rule

$$
\begin{aligned}
\frac{dy}{dx} &= \frac{dy}{dw_{n-1}} \frac{dw_{n-1}}{dx} \\
&= \frac{dy}{dw_{n-1}} \left(\frac{dw_{n-1}}{dw_{n-2}} \frac{dw_{n-2}}{dx} \right) \\
&= \frac{dy}{dw_{n-1}} \left(\frac{dw_{n-1}}{dw_{n-2}} \left(\frac{dw_{n-2}}{dw_{n-3}} \frac{dw_{n-3}}{dx} \right) \right)
\end{aligned}
$$

- We used the chain rule following the steps of execution, as in derivative information flowed alongside the primals

Another formulation

- We can move around parenthesis because multiplication is associative

$$
\begin{aligned}
\frac{dy}{dx} &= \frac{dy}{dw_{n-1}}\left(\frac{dw_{n-1}}{dw_{n-2}}\left(\frac{dw_{n-2}}{dw_{n-3}}\frac{dw_{n-3}}{dx}\right)\right) \\
&= \left(\left(\frac{dy}{dw_{n-1}}\frac{dw_{n-1}}{dw_{n-2}}\right)\frac{dw_{n-2}}{dw_{n-3}}\right)\frac{dw_{n-3}}{dx}
\end{aligned}
$$

- The first product we need to compute is the derivative of the output (dependent variable y) with respects to its sub expressions
- This implies we need to know the primal values and what functions created them when we get to the end of the graph
- Then, we have to walk the graph backwards to accumulate these products
- This is "reverse mode" AD

Reverse Mode Example

$$y = f(g(x)) \tag{5}$$

$$\boxed{\text{x}}$$

$$p_1 = x \qquad a_1 = a_2 \frac{\partial p_2}{\partial p_1} = a_2 g'(p_1)$$

g

$$p_2 = g(p_1) \qquad a_2 = a_3 \frac{\partial p_3}{\partial p_2} = a_3 f'(p_2)$$

f

$$p_3 = f(p_2) \qquad a_3 = 1$$

$$\boxed{\text{y}}$$

Implementation Strategies / Ecosystems

- There are a few approaches to *both* styles of AD
    - Operator overloading
        - Easiest to implement, limited applications as functions have to be generic
    - Source transformation
        - Hardest to implement, but should work on all programs
- Most languages have decent libraries
    - Python
        - JAX
        - PyTorch
    - Julia
        - Enzyme
        - ForwardDiff/ReverseDiff
        - Zygote
    - C/C++
        - Enzyme
        - autodiff
        - stan
        - Adept

Demos!

Demo time