Ch 1 Auto Differentiation (AD) aka "Algorithmic Differentiation"

Monday, September 16, 2024 10:11 AM

"Reverse mode" is aka "Backpropagation"

contested history, rediscovered many times

See associated Jupyter notebook

> "Under quite reasonable assumptions, the evaluation of the gradient requires no more than 5x the effort of evaluating the underlying function itself"
> — Andreas Griewank '88

Software Examples

ADIFOR — Fortran code, won Wilkinson prize in 1995

TensorFlow, PyTorch, Jax — modern Python packages

Adimat — Matlab code (not a lot of good options in Matlab)

... many ... — Julia has many good options

What is AD?

Not finite differences. Not exactly symbolic. It's using the chain rule (like symbolic) but in computationally efficient ways.

Applies automatically to your source code, and some AD packages can handle "for" loops and other complicated constructions

but first, motivation

If you can compute derivative symbolically, should you?

Ex Speed penning function $g(x) = \prod_{i=1}^{m} g_i(x)$, eg take $g_i(x) = x - t_i$, $x \in \mathbb{R}$

so $g(x) = \prod_{i=1}^{m} (x - t_i)$, i.e., a m-degree polynomial in factored form

A polynomial in coefficient form $a_n x^n + a_{n-1} x^{n-1} + \dots$ can be efficiently evaluated via Horner's method, and also easy to find derivative

but in this factored form, a naive symbolic differentiator might give you the Calc I product rule output:

$$g'(x) = \sum_{i=1}^{m} \prod_{j \neq i} (x - t_j)$$ which takes $O(m^2)$ to evaluate!

Skip in lecture

or $$g'(x) = \sum_{i=1}^{m} \frac{g(x)}{x - t_i}$$ now $O(m)$ but unstable when $x \approx t_i$

Reverse-Mode AD approach speeds this up (and not unstable) at the cost of extra memory:

let $f^{(0)} = 1$, $f^{(K)} = \prod_{i=1}^{K} x - t_i$ "forward"

$r^{(K)} = \prod_{i=K}^{n} x - t_i$, $r^{(n+1)} = 1$ "reverse"

```
def g(x):
    f = 1
    for i in range(m):
        f = f * (x - t[i])
    return f
```

computing this is cheap ($O(m)$) if done like

$f^{(K)} = (x - t_K) f^{(K-1)}$

$r^{(K)} = (x - t_K) r^{(K+1)}$

... but adds $O(m)$ storage ...

Fundamental speed vs. storage tradeoff in AD

then $$g'(x) = \sum_{i=1}^{m} f^{(i-1)} \cdot r^{(i+1)}$$

Symbolic vs Automatic Differentiation

Monday, September 16, 2024 10:31 AM

Symbolic Differentiation

Gives a mathematical object

Doesn't give an implementation

(or, in practice, it does give you one implementation.
Need not be a good one.
You might be able to call Mathematica's "simplify"
to get another)

i.e. an equivalence class of implementations

ex: $f(x) = \sin(x)$

$f'(x) = \cos(x) = \cos(x) + (7 - 7 = \sin(\pi/2 - x)$

$= \frac{\sin(x)}{\tan(x)} = \ldots$ many implementations!

AD

Computes derivative, but how depends not just on the function, but depends on how you implement it.

ex: $f(x) = 0$

```
def f(x):
    return x - x
```

then AD will (implicitly) define

```
def fprime(x):
    return 1 - 1
```

Mathematical and computational object

(In-class exercise: what's the complexity, in big-O notation, of computing

① $A \cdot \vec{x}$ (A: $n \times n$ matrix, $\vec{x}$: $n \times 1$ vector)

② $A \cdot B$ (both $n \times n$ matrices)

Answer: ① $O(n^2)$, ② $O(n^3)$ (also discussed Strassen...)

Forward vs Reverse Mode

Thursday, September 4, 2025 5:00 PM

AutoDiff is efficiently applying the chain rule

Write your function as a composition, sum and/or product of simpler functions... until it's in terms of simple primitives (like sin(·), multiplication, e^x, etc). for which we know the derivatives.

(conceptually... in practice, the software does this)

Simplest explanation is via an example:

$f: \mathbb{R}^n \to \mathbb{R}$, $f(\vec{x}) = \text{sum}(B \cdot (A \cdot \vec{x}))$, $A, B \in \mathbb{R}^{n \times n}$, $\vec{x} \in \mathbb{R}^n$

$= \underbrace{\mathbb{1}^T \cdot B \cdot A}_{\vec{c}^T} \cdot \vec{x}$, $\mathbb{1} \in \mathbb{R}^n$ is all ones vector.

$= \vec{c}^T \cdot \vec{x}$.

Fact: for a linear function $f: \mathbb{R}^n \to \mathbb{R}^m$, $f(\vec{x}) = C \cdot \vec{x}$, $C \in \mathbb{R}^{m \times n}$

then $J_f(\vec{x}) = C$

so $J_f(\vec{x}) = \vec{c}^T$, i.e. $\nabla f(\vec{x}) = \vec{c}$ $(= A^T B^T \cdot \mathbb{1})$. } our derivation by hand

What does AutoDiff do?

$f(\vec{x}) = p(q(r(\vec{x})))$, (primitives; known gradients)

$r(\vec{x}) = A \cdot \vec{x}$, $J_r(\vec{x}) = A$

So

$q(\vec{y}) = B \cdot \vec{y}$, $J_q(\vec{y}) = B$

$p(\vec{z}) = \mathbb{1}^T \cdot \vec{z}$, $J_p(\vec{z}) = \mathbb{1}^T$

$J_f(\vec{x}) = J_p(q(r(\vec{x}))) \cdot J_q(r(\vec{x})) \cdot J_r(\vec{x})$ } Chain rule

to evaluate this, two options:

① FORWARD MODE

$J_f(\vec{x}) = \left(J_p(q(r(\vec{x}))) \cdot \left(J_q(r(\vec{x})) \cdot \left(J_r(\vec{x})\right)\right)\right)$ (③, ②, ①)

$= \mathbb{1}^T \cdot (B \cdot A)$

Cost: $B \cdot A$ is $O(n^3)$ flops

$\mathbb{1}^T(BA)$ is $O(n^2)$

... so $O(n^3)$

FACT: multiplying 2 $n \times n$ matrices the standard way costs $O(n^3)$ flops

② REVERSE MODE

$J_f(\vec{x}) = \left(\left(J_p(q(r(\vec{x})))\right) \cdot J_q(r(\vec{x}))\right) \cdot J_r(\vec{x})$ (①, ②, ③)

$= (\mathbb{1}^T \cdot B) \cdot A$

Cost: $\mathbb{1}^T B$ is $O(n^2)$ flops

$(\mathbb{1}^T B) \cdot A$ is $O(n^2)$ ($1 \times n$, $n \times n$)

... so $O(n^2)$

Catch! what is this value?

Need to do a forward pass first, save value, then make backward pass. REQUIRES A LOT OF MEMORY

More detail on next page

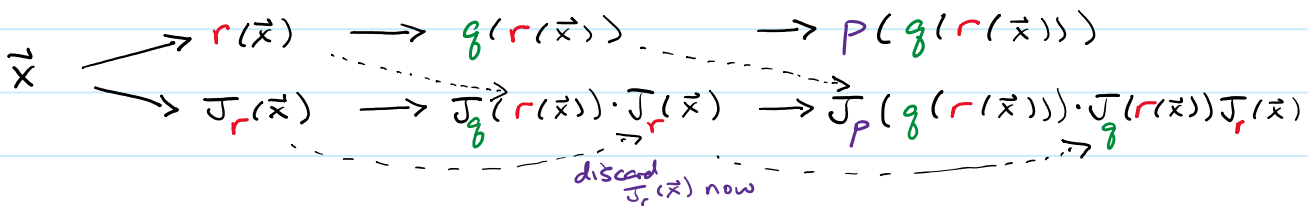
AD in more detail

Friday, September 5, 2025 9:32 AM

... in more detail:

FORWARD MODE $f(\vec{x}) = p(q(r(\vec{x})))$

$$J_f(\vec{x}) = \left(J_p(q(r(\vec{x}))) \cdot \left(J_q(r(\vec{x})) \cdot \left(J_r(\vec{x})\right)\right)\right)$$



$B \cdot A$

$(n \times n) \cdot (n \times n)$ is $O(n^3)$ flops

$\mathbb{1}^T (BA)$ $(1 \times n)(n \times n)$ is $O(n^2)$ flops ... $O(n^3)$ flops total

REVERSE MODE

Forward pass $\vec{x} \to$ SAVE $r(\vec{x}) \to$ SAVE $q(r(\vec{x})) \to p(q(r(\vec{x})))$

$J_p(q(r(\vec{x})))J_q(r(\vec{x}))J_r(\vec{x}) \leftarrow J_p(q(r(\vec{x})))J_q(r(\vec{x})) \leftarrow J_p(q(r(\vec{x})))$

more efficient in terms of computation (if $f: \mathbb{R}^n \to \mathbb{R}$)

but requires more memory

In general, if

* $f: \mathbb{R}^n \to \mathbb{R}^1$ (common setting)

forward-mode costs n times the cost of a function evaluation

* reverse-mode costs the same as a function eval.

$f: \mathbb{R}^1 \to \mathbb{R}^m$

forward-mode costs the same as a function evaluation

reverse-mode costs m times...

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

forward mode costs n times the cost of a function eval

reverse-mode costs m times ...

"Backprop" = Reverse-mode, popular since $f: \mathbb{R}^n \to \mathbb{R}$ is common

... but memory issues!

Hessian Vector Product "Hvp"

is $\nabla^2 f(\vec{x}) \cdot \vec{v}$ for some $\vec{v}$. Can be done efficiently (do AD on gradient via AD)

Useful for Taylor series, matrix-free linear sys. solves. $\nabla^2 f(\vec{x})\vec{v} = \frac{d}{dt}\varphi(t)\Big|_{t=0}$, $\varphi(t) = \nabla f(\vec{x} + t\vec{v})$, $\varphi: \mathbb{R}^1 \to \mathbb{R}^n$