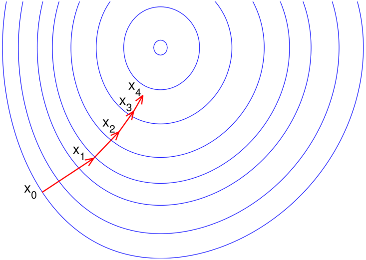
Basics of Numerical Optimization: Computing Derivatives

Ju Sun
Computer Science & Engineering
University of Minnesota, Twin Cities

September 28, 2023

Derivatives for numerical optimization



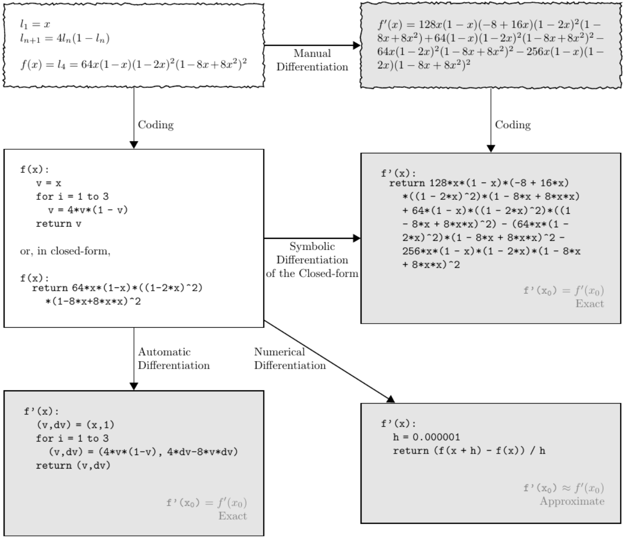
Credit: aria42.com

- gradient descent
- Newton's method
- momentum methods
- quasi-Newton methods
- coordinate descent
- conjugate gradient methods
- trust-region methods

- Almost all methods entail low-order derivatives, i.e., gradient and/or Hessian, to proceed.
  - 1st order methods: use $f(\boldsymbol{x})$ and $\nabla f(\boldsymbol{x})$
  - 2nd order methods: use $f(\boldsymbol{x})$ and $\nabla f(\boldsymbol{x})$ and $\nabla^2 f(\boldsymbol{x})$
- **Numerical (not analytical) derivatives** (i.e., numbers) needed for the iterations

This lecture: how to compute the numerical derivatives

Four kinds of computing techniques



Credit: [Baydin et al., 2017]

Outline

Analytical differentiation

Finite-difference approximation

Automatic differentiation

Differentiable programming

Suggested reading

Analytical derivatives

Idea: derive the analytical derivatives first, then make numerical substitution

To derive the analytical derivatives by hand:

- **Chain rule (vector version) method**
  Let $f : \mathbb{R}^m \to \mathbb{R}^n$ and $h : \mathbb{R}^n \to \mathbb{R}^k$, and f is differentiable at $\boldsymbol{x}$ and $\boldsymbol{z} = h(\boldsymbol{y})$ is differentiable at $\boldsymbol{y} = f(\boldsymbol{x})$. Then, $\boldsymbol{z} = h \circ f(\boldsymbol{x}) : \mathbb{R}^m \to \mathbb{R}^k$ is differentiable at $\boldsymbol{x}$, and

$$\boldsymbol{J}_{[h \circ f]}(\boldsymbol{x}) = \boldsymbol{J}_h(f(\boldsymbol{x}))\, \boldsymbol{J}_f(\boldsymbol{x}), \text{ or } \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

  When $k = 1$**,**

$$\nabla [h \circ f](\boldsymbol{x}) = \boldsymbol{J}_f^\top(\boldsymbol{x})\, \nabla h(f(\boldsymbol{x})).$$

- **Taylor expansion method**
  Expand the perturbed function $f(\boldsymbol{x} + \boldsymbol{\delta})$ and then match it against Taylor expansions to read off the gradient and/or Hessian:

$$f(\boldsymbol{x} + \boldsymbol{\delta}) = f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{\delta} \rangle + o\left(\|\boldsymbol{\delta}\|_2\right)$$

$$f(\boldsymbol{x} + \boldsymbol{\delta}) = f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{\delta} \rangle + \frac{1}{2} \left\langle \boldsymbol{\delta}, \nabla^2 f(\boldsymbol{x})\, \boldsymbol{\delta} \right\rangle + o\left(\|\boldsymbol{\delta}\|_2^2\right)$$

Symbolic differentiation

Idea: derive the analytical derivatives first, then make numerical substitution

To derive the analytical derivatives by software:

Differentiate Function

Find the derivative of the function `sin(x^2)`.

```
syms f(x)
f(x) = sin(x^2);
df = diff(f,x)
```

```
df(x) =
2*x*cos(x^2)
```

Find the value of the derivative at `x = 2`. Convert the value to double.

```
df2 = df(2)
```

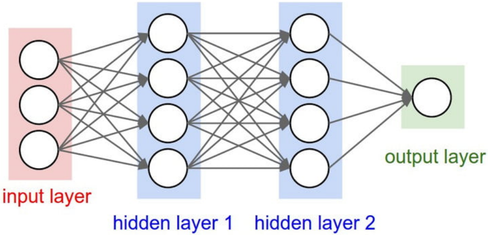
```
df2 =
4*cos(4)
```

- Matlab (Symbolic Math Toolbox, `diff`)
- Python (SymPy, `diff`)
- Mathmatica (`D`)
- Matrix Calculus https://www.matrixcalculus.org/

Effective for simple functions

Outline

Limitation of analytical differentiation



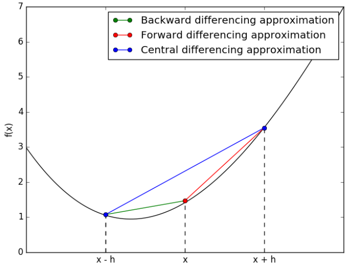
What is the gradient and/or Hessian of

$$f\left(\boldsymbol{W}\right) = \sum_i \left\|\boldsymbol{y}_i - \sigma\left(\boldsymbol{W}_k \sigma\left(\boldsymbol{W}_{k-1}\sigma \dots \left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)\right\|_F^2?$$

Applying the chain rule is boring and error-prone. Performing Taylor expansion can also be tedious

Lesson we learn from tech history: leave boring jobs to computers

Approximate the gradient



(Credit: numex-blog.com)

$f'(\boldsymbol{x}) = \lim_{\delta \to 0} \frac{f(x+\delta)-f(x)}{\delta} \approx \frac{f(x+\delta)-f(x)}{\delta}$
with δ sufficiently small

For $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$,

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\boldsymbol{x}+\delta \boldsymbol{e}_i) - f(\boldsymbol{x})}{\delta} \quad \text{(forward)}$$

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\boldsymbol{x}) - f(\boldsymbol{x}-\delta \boldsymbol{e}_i)}{\delta} \quad \text{(backward)}$$

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\boldsymbol{x}+\delta \boldsymbol{e}_i) - f(\boldsymbol{x}-\delta \boldsymbol{e}_i)}{2\delta} \quad \text{(central)}$$

Similarly, to approximate the Jacobian for $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$:

$$\frac{\partial f_j}{\partial x_i} \approx \frac{f_j(\boldsymbol{x}+\delta \boldsymbol{e}_i) - f_j(\boldsymbol{x})}{\delta} \quad \text{(one element each time)}$$

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\boldsymbol{x}+\delta \boldsymbol{e}_i) - f(\boldsymbol{x})}{\delta} \quad \text{(one column each time)}$$

$$\boldsymbol{J}_f(\boldsymbol{x})\boldsymbol{p} \approx \frac{f(\boldsymbol{x}+\delta \boldsymbol{p}) - f(\boldsymbol{x})}{\delta} \quad \text{(directional)}$$

central themes can also be derived

Why central?

Stronger form of Taylor's theorems

- **1st order**: If $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable,
  $f(\boldsymbol{x}+\boldsymbol{\delta}) = f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{\delta} \rangle + O\left(\|\boldsymbol{\delta}\|_2^2\right)$
- **2nd order**: If $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$ is three-times continuously differentiable,
  $f(\boldsymbol{x}+\boldsymbol{\delta}) = f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{\delta} \rangle + \frac{1}{2}\left\langle \boldsymbol{\delta}, \nabla^2 f(\boldsymbol{x}) \boldsymbol{\delta} \right\rangle + O\left(\|\boldsymbol{\delta}\|_2^3\right)$

Why the central theme is better?

- Forward: by 1st-order Taylor expansion
  $\frac{1}{\delta}\left(f(\boldsymbol{x}+\delta \boldsymbol{e}_i) - f(\boldsymbol{x})\right) = \frac{1}{\delta}\left(\delta \frac{\partial f}{\partial x_i} + O\left(\delta^2\right)\right) = \frac{\partial f}{\partial x_i} + O(\delta)$
- Central: by 2nd-order Taylor expansion $\frac{1}{\delta}\left(f(\boldsymbol{x}+\delta \boldsymbol{e}_i) - f(\boldsymbol{x}-\delta \boldsymbol{e}_i)\right) =$
  $\frac{1}{2\delta}\left(\delta \frac{\partial f}{\partial x_i} + \frac{1}{2}\delta^2 \frac{\partial^2 f}{\partial x_i^2} + \delta \frac{\partial f}{\partial x_i} - \frac{1}{2}\delta^2 \frac{\partial^2 f}{\partial x_i^2} + O\left(\delta^3\right)\right) = \frac{\partial f}{\partial x_i} + O(\delta^2)$

Approximate the Hessian

- Recall that for $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$ that is 2nd-order differentiable, $\frac{\partial f}{\partial x_i}(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$. So

$$\frac{\partial f^2}{\partial x_j \partial x_i}(\boldsymbol{x}) = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}\right)(\boldsymbol{x}) \approx \frac{\left(\frac{\partial f}{\partial x_i}\right)(\boldsymbol{x} + \delta \boldsymbol{e}_j) - \left(\frac{\partial f}{\partial x_i}\right)(\boldsymbol{x})}{\delta}$$

- We can also compute one row of Hessian each time by

$$\frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial \boldsymbol{x}}\right)(\boldsymbol{x}) \approx \frac{\left(\frac{\partial f}{\partial \boldsymbol{x}}\right)(\boldsymbol{x} + \delta \boldsymbol{e}_j) - \left(\frac{\partial f}{\partial \boldsymbol{x}}\right)(\boldsymbol{x})}{\delta},$$

  obtaining $\widehat{H}$, which might not be symmetric. Return $\frac{1}{2}\left(\widehat{\boldsymbol{H}} + \widehat{\boldsymbol{H}}^\intercal\right)$ instead

- Most times (e.g., in TRM, Newton-CG), only $\nabla^2 f(\boldsymbol{x})\, \boldsymbol{v}$ for certain $\boldsymbol{v}$'s needed: (see, e.g., Manopt https://www.manopt.org/)

$$\nabla^2 f(\boldsymbol{x})\, \boldsymbol{v} \approx \frac{\nabla f(\boldsymbol{x} + \delta \boldsymbol{v}) - \nabla f(\boldsymbol{x})}{\delta}$$

A few words

- Can be used for sanity check of correctness of analytical gradient
- Finite-difference approximation of higher (i.e., ≥ 2)-order derivatives combined with high-order iterative methods can be very efficient (e.g., Manopt https://www.manopt.org/tutorial.html#costdescription)
- Numerical stability can be an issue: truncation and round off errors (finite δ; accurate evaluation of the nominators)

Outline

Analytical differentiation

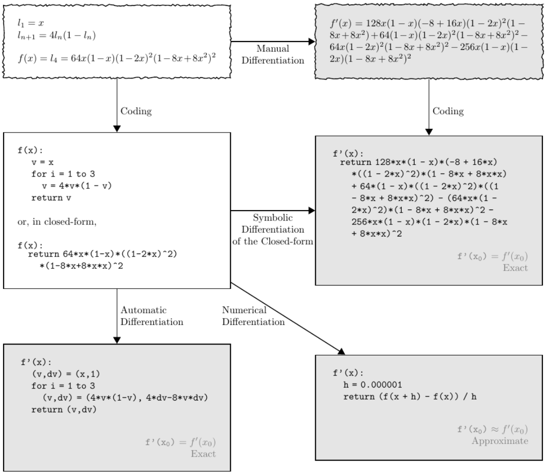
Finite-difference approximation

Automatic differentiation

Differentiable programming

Suggested reading

Four kinds of computing techniques

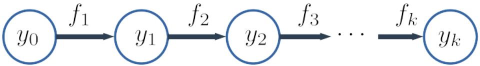


Credit: [Baydin et al., 2017]

Misnomer: should be **automatic numerical differentiation**

Auto differentiation (auto diff, AD) in 1D

Consider a univariate function $f_k \circ f_{k-1} \circ \cdots \circ f_2 \circ f_1 (x) : \mathbb{R} \to \mathbb{R}$. Write $y_0 = x$, $y_1 = f_1 (x)$, $y_2 = f_2 (y_1)$, $\ldots$, $y_k = f (y_{k-1})$, or in **computational graph** form:



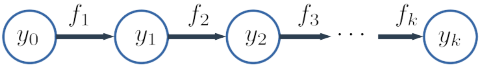
Chain rule in Leibniz form:

$$\frac{\partial f}{\partial x} = \frac{\partial y_k}{\partial y_0} = \frac{\partial y_k}{\partial y_{k-1}} \frac{\partial y_{k-1}}{\partial y_{k-2}} \cdots \frac{\partial y_2}{\partial y_1} \frac{\partial y_1}{\partial y_0}$$

How to evalute the product?

- From left to right in the chain: **forward mode auto diff**
- From right to left in the chain: **backward/reverse mode auto diff**
- Hybrid: mixed mode

Forward mode in 1D



Chain rule: $\dfrac{df}{dx} = \dfrac{dy_k}{dy_0} = \left(\dfrac{dy_k}{dy_{k-1}} \left(\dfrac{dy_{k-1}}{dy_{k-2}} \left(\dots \left(\dfrac{dy_2}{dy_1} \left(\dfrac{dy_1}{dy_0} \right) \right) \right) \right) \right)$

Example: For $f(\boldsymbol{x}) = \left(x^2 + 1\right)^2$, calculate $\nabla f(1)$ (whiteboard)

Compute $\left.\frac{df}{dx}\right|_{x_0}$ in one pass, from inner to outer most parenthesis:

Input: y_0, initialization $\left.\frac{dy_0}{dy_0}\right|_{y_0} = 1$

for $i = 1, \dots, k$ **do**

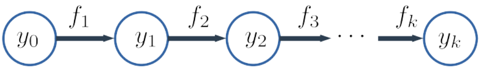
compute $y_i = f_i(y_{i-1})$

compute $\left.\frac{dy_i}{dy_0}\right|_{y_0} = \left.\frac{dy_i}{dy_{i-1}}\right|_{y_{i-1}} \cdot \left.\frac{dy_{i-1}}{dy_0}\right|_{y_0} = f_i'(y_{i-1}) \left.\frac{dy_{i-1}}{dy_0}\right|_{y_0}$

end for

Output: $\left.\frac{dy_k}{dy_0}\right|_{y_0}$

Reverse mode in 1D



Chain rule: $$\frac{df}{dx} = \frac{df}{dy_0} = \left(\left(\left(\left(\left(\frac{dy_k}{dy_{k-1}}\right)\frac{dy_{k-1}}{dy_{k-2}}\right)\dots\right)\frac{dy_2}{dy_1}\right)\frac{dy_1}{dy_0}\right)$$

Example: For $f(\boldsymbol{x}) = \left(x^2+1\right)^2$, calculate $\nabla f(1)$ (whiteboard)

Compute $\frac{df}{dx}\Big|_{x_0}$ in two passes:

- Forward pass: calculate the y_i's sequentially
- Backward pass: calculate the $\frac{dy_k}{dy_i} = \frac{dy_k}{dy_{i+1}}\frac{dy_{i+1}}{dy_i}$ backward

---

Input: y_0, $\frac{dy_k}{dy_k} = 1$

for $i = 1, \dots, k$ **do**

compute $y_i = f_i(y_{i-1})$

end for // forward pass

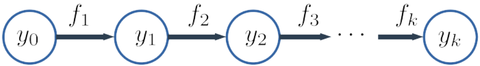
for $i = k-1, k-2, \dots, 0$ **do**

compute $\frac{dy_k}{dy_i}\Big|_{y_i} = \frac{dy_k}{dy_{i+1}}\Big|_{y_{i+1}} \cdot \frac{dy_{i+1}}{dy_i}\Big|_{y_i} = f'_{i+1}(y_i)\,\frac{dy_k}{dy_{i+1}}\Big|_{y_{i+1}}$

end for // backward pass

Output: $\frac{dy_k}{dy_0}\Big|_{y_0}$

Forward vs reverse modes



- **forward mode AD**: one forward pass, compute y_i's and $\frac{dy_i}{dy_0}$'s together
- **reverse mode AD**: one forward pass to compute y_i's, one backward pass to compute $\frac{dy_k}{dy_i}$'s

Effectively, two different ways of grouping the multiplicative differential terms:

$$\frac{df}{dx} = \frac{df}{dy_0} = \left(\frac{dy_k}{dy_{k-1}}\left(\frac{dy_{k-1}}{dy_{k-2}}\left(\cdots\left(\frac{dy_2}{dy_1}\left(\frac{dy_1}{dy_0}\right)\right)\right)\right)\right)$$

i.e., starting from the root: $\frac{dy_0}{dy_0} \mapsto \frac{dy_1}{dy_0} \mapsto \frac{dy_2}{dy_0} \mapsto \cdots \mapsto \frac{dy_k}{dy_0}$

$$\frac{df}{dx} = \frac{df}{dy_0} = \left(\left(\left(\left(\left(\frac{dy_k}{dy_{k-1}}\right)\frac{dy_{k-1}}{dy_{k-2}}\right)\cdots\right)\frac{dy_2}{dy_1}\right)\frac{dy_1}{dy_0}\right)$$

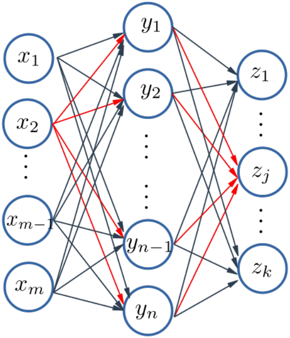
i.e., starting from the leaf: $\frac{dy_k}{dy_k} \mapsto \frac{dy_k}{dy_{k-1}} \mapsto \frac{dy_k}{dy_{k-2}} \mapsto \cdots \mapsto \frac{dy_k}{dy_0}$

...mixed forward and reverse modes are indeed possible!

Auto differentiation in high dimensions

Chain Rule Let $f : \mathbb{R}^m \to \mathbb{R}^n$ and $h : \mathbb{R}^n \to \mathbb{R}^k$, and f is differentiable at $\boldsymbol{x}$ and $\boldsymbol{z} = h(\boldsymbol{y})$ is differentiable at $\boldsymbol{y} = f(\boldsymbol{x})$. Then, $\boldsymbol{z} = h \circ f(\boldsymbol{x}) : \mathbb{R}^m \to \mathbb{R}^k$ is differentiable at $\boldsymbol{x}$, and

$$\boldsymbol{J}_{[h \circ f]}(\boldsymbol{x}) = \boldsymbol{J}_h(f(\boldsymbol{x}))\,\boldsymbol{J}_f(\boldsymbol{x}), \text{ or } \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \Leftrightarrow \frac{\partial z_j}{\partial x_i} = \sum_{\ell=1}^{n} \frac{\partial z_j}{\partial y_\ell} \frac{\partial y_\ell}{\partial x_i} \; \forall\, i, j$$

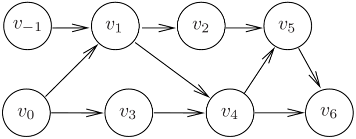


NB: this is a computational graph, not a NN

- Each node is a variable, as a function of all incoming variables
- If node B a child of node A, $\frac{\partial B}{\partial A}$ is the rate of change in B wrt change in A
- Traveling along a path, rates of changes should be multiplied
- Chain rule: summing up rates over all connecting paths! (e.g., x_2 to z_j as shown)

A multivariate example—forward mode

$$y = \left(\sin\frac{x_1}{x_2} + \frac{x_1}{x_2} - e^{x_2}\right)\left(\frac{x_1}{x_2} - e^{x_2}\right)$$

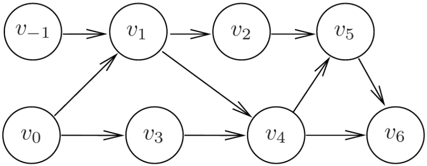


v_{-1}	$=$	x_1	$=$	1.5000		
v_0	$=$	x_2	$=$	0.5000		
v_1	$=$	v_{-1}/v_0	$=$	$1.5000/0.5000$	$=$	3.0000
v_2	$=$	$\sin(v_1)$	$=$	$\sin(3.0000)$	$=$	0.1411
v_3	$=$	$\exp(v_0)$	$=$	$\exp(0.5000)$	$=$	1.6487
v_4	$=$	$v_1 - v_3$	$=$	$3.0000 - 1.6487$	$=$	1.3513
v_5	$=$	$v_2 + v_4$	$=$	$0.1411 + 1.3513$	$=$	1.4924
v_6	$=$	$v_5 * v_4$	$=$	$1.4924 * 1.3513$	$=$	2.0167
y	$=$	v_6	$=$	2.0167		

$v_{-1} = x_1$	$= 1.5000$		
$\dot{v}_{-1} = \dot{x}_1$	$= 1.0000$		
$v_0 = x_2$	$= 0.5000$		
$\dot{v}_0 = \dot{x}_2$	$= 0.0000$		
$v_1 = v_{-1}/v_0$	$= 1.5000/0.5000$	$=$	3.0000
$\dot{v}_1 = (\dot{v}_{-1} - v_1 * \dot{v}_0)/v_0$	$= 1.0000/0.5000$	$=$	2.0000
$v_2 = \sin(v_1)$	$= \sin(3.0000)$	$=$	0.1411
$\dot{v}_2 = \cos(v_1) * \dot{v}_1$	$= -0.9900 * 2.0000$	$=$	-1.9800
$v_3 = \exp(v_0)$	$= \exp(0.5000)$	$=$	1.6487
$\dot{v}_3 = v_3 * \dot{v}_0$	$= 1.6487 * 0.0000$	$=$	0.0000
$v_4 = v_1 - v_3$	$= 3.0000 - 1.6487$	$=$	1.3513
$\dot{v}_4 = \dot{v}_1 - \dot{v}_3$	$= 2.0000 - 0.0000$	$=$	2.0000
$v_5 = v_2 + v_4$	$= 0.1411 + 1.3513$	$=$	1.4924
$\dot{v}_5 = \dot{v}_2 + \dot{v}_4$	$= -1.9800 + 2.0000$	$=$	0.0200
$v_6 = v_5 * v_4$	$= 1.4924 * 1.3513$	$=$	2.0167
$\dot{v}_6 = \dot{v}_5 * v_4 + v_5 * \dot{v}_4$	$= 0.0200 * 1.3513 + 1.4924 * 2.0000$	$=$	3.0118
$y = v_6$	$= 2.0100$		
$\dot{y} = \dot{v}_6$	$= 3.0110$		

- interested in $\frac{\partial}{\partial x_1}$; for each variable v_i, write $\dot{v}_i \doteq \frac{\partial v_i}{\partial x_1}$
- for each node, sum up partials over all incoming edges, e.g., $\dot{v}_4 = \frac{\partial v_4}{\partial v_1}\dot{v}_1 + \frac{\partial v_4}{\partial v_3}\dot{v}_3$
- complexity: $O(\#\text{edges} + \#\text{nodes})$
- for $f : \mathbb{R}^n \to \mathbb{R}^m$, make n forward passes: $O(n(\#\text{edges} + \#\text{nodes}))$

A multivariate example—reverse mode



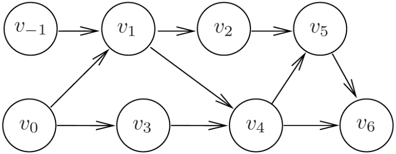
$$
\begin{aligned}
v_{-1} &= x_1 = 1.5000 \\
v_0 &= x_2 = 0.5000 \\
v_1 &= v_{-1}/v_0 = 1.5000/0.5000 = 3.0000 \\
v_2 &= \sin(v_1) = \sin(3.0000) = 0.1411 \\
v_3 &= \exp(v_0) = \exp(0.5000) = 1.6487 \\
v_4 &= v_1 - v_3 = 3.0000 - 1.6487 = 1.3513 \\
v_5 &= v_2 + v_4 = 0.1411 + 1.3513 = 1.4924 \\
v_6 &= v_5 * v_4 = 1.4924 * 1.3513 = 2.0167 \\
y &= v_6 = 2.0167 \\
\bar{v}_6 &= \bar{y} = 1.0000 \\
\bar{v}_5 &= \bar{v}_6 * v_4 = 1.0000 * 1.3513 = 1.3513 \\
\bar{v}_4 &= \bar{v}_6 * v_5 = 1.0000 * 1.4924 = 1.4924 \\
\bar{v}_4 &= \bar{v}_4 + \bar{v}_5 = 1.4924 + 1.3513 = 2.8437 \\
\bar{v}_2 &= \bar{v}_5 = 1.3513 \\
\bar{v}_3 &= -\bar{v}_4 = -2.8437 \\
\bar{v}_1 &= \bar{v}_4 = 2.8437 \\
\bar{v}_0 &= \bar{v}_3 * v_3 = -2.8437 * 1.6487 = -4.6884 \\
\bar{v}_1 &= \bar{v}_1 + \bar{v}_2 * \cos(v_1) = 2.8437 + 1.3513 * (-0.9900) = 1.5059 \\
\bar{v}_0 &= \bar{v}_0 - \bar{v}_1 * v_1/v_0 = -4.6884 - 1.5059 * 3.000/0.5000 = -13.7239 \\
\bar{v}_{-1} &= \bar{v}_1/v_0 = 1.5059/0.5000 = 3.0118 \\
\bar{x}_2 &= \bar{v}_0 = -13.7239 \\
\bar{x}_1 &= \bar{v}_{-1} = 3.0118
\end{aligned}
$$

- interested in $\frac{\partial y}{\partial}$; for each variable v_i, write $\overline{v}_i \doteq \frac{\partial y}{\partial v_i}$ (called **adjoint variable**)
- for each node, sum up partials over all outgoing edges, e.g., $\overline{v}_4 = \frac{\partial v_5}{\partial v_4}\overline{v}_5 + \frac{\partial v_6}{\partial v_4}\overline{v}_6$
- complexity: $O\left(\#\text{edges} + \#\text{nodes}\right)$
- for $f : \mathbb{R}^n \to \mathbb{R}^m$, make m backward passes: $O\left(m\left(\#\text{edges} + \#\text{nodes}\right)\right)$

example from Ch 1 of [Griewank and Walther, 2008]

Forward vs. reverse modes

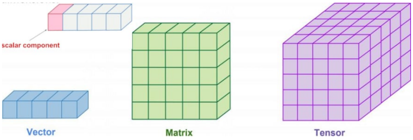
For general function $f : \mathbb{R}^n \to \mathbb{R}^m$, suppose there is no loop in the computational graph, i.e., **acyclic graph**. E: set of edges ; V: set of nodes



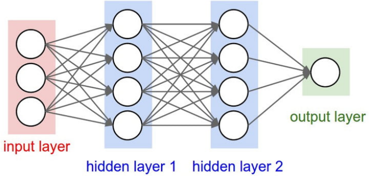
	forward mode	**reverse mode**
start from	roots	leaves
end with	leaves	roots
invariants	$\dot{v}_i \doteq \frac{\partial v_i}{\partial x}$ (x—root of interest)	$\overline{v}_i \doteq \frac{\partial y}{\partial v_i}$ (y—leaf of interest)
rule	sum over incoming edges	sum over outgoing edges
computation	$O(n\,\lvert E\rvert + n\,\lvert V\rvert)$	$O(m\,\lvert E\rvert + m\,\lvert V\rvert)$
memory	$O(\lvert V\rvert)$, typically way smaller	$O(\lvert V\rvert)$
better when	$m \gg n$	$n \gg m$

Implementation trick—tensor abstraction

Tensors: multi-dimensional arrays

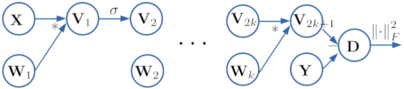


Each node in the computational graph can be a tensor (scalar, vector, matrix, 3-D tensor, ...)

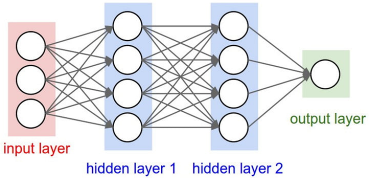


$$f(\boldsymbol{W}) = \|\boldsymbol{Y} - \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \sigma \dots (\boldsymbol{W}_1 \boldsymbol{X})))\|_F^2$$

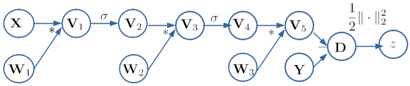
computational graph for DNN



Implementation trick—tensor abstraction



computational graph for DNN



$$f(\boldsymbol{W}) = \|\boldsymbol{Y} - \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1}\sigma \dots (\boldsymbol{W}_1 \boldsymbol{X})))\|_F^2$$

- neater computational graph
- tensor (i.e., vector) chain rules apply, often in tensor-free computation
  Fact: For two matrices (tensors) $\boldsymbol{D}$ and $\boldsymbol{M}$ of compatiable size, where $\boldsymbol{D}$ is fixed and $\boldsymbol{M}$ is a function of $\boldsymbol{M}'$

  $$\nabla_{\boldsymbol{M}'} \langle \boldsymbol{M}, \boldsymbol{D} \rangle = \mathcal{J}^{\intercal}_{\boldsymbol{M}' \to \boldsymbol{M}}(\boldsymbol{M}')[\boldsymbol{D}]$$

  * EX1: $\frac{\partial f}{\partial \boldsymbol{V}_4}$ (whiteboard)
  * EX2: $\frac{\partial f}{\partial \boldsymbol{V}_1}$ (whiteboard)

Implementation trick—VJP

Interested in $\boldsymbol{J}_f(\boldsymbol{x})$ for $f : \mathbb{R}^n \mapsto \mathbb{R}^m$. Implement $\boldsymbol{v}^\intercal \boldsymbol{J}_f(\boldsymbol{x})$ for any $\boldsymbol{v} \in \mathbb{R}^m$

- Why?
  * set $\boldsymbol{v} = e_i$ for $i = 1, \dots, m$ to recover rows of $\boldsymbol{J}_f(\boldsymbol{x})$
  * special structures in $\boldsymbol{J}_f(\boldsymbol{x})$ (e.g., sparsity) can be exploited
  * often enough for application, e.g., calculate $\nabla(g \circ f) = (\nabla f^\intercal \boldsymbol{J}_f)^\intercal$ with known ∇f
- Why possible?
  * $\boldsymbol{v}^\intercal \boldsymbol{J}_f(\boldsymbol{x}) = \boldsymbol{J}_{\boldsymbol{v}^\intercal f}(\boldsymbol{x})$ so keep track of $\frac{\partial}{\partial v_i}(\boldsymbol{v}^\intercal f) = \sum_{k:\text{outgoing}} \frac{\partial v_k}{\partial v_i} \frac{\partial}{\partial v_k}(\boldsymbol{v}^\intercal f)$
  * implemeted in reverse-mode auto diff

```
torch.autograd.functional.vjp(func, inputs, v=None, create_graph=False, strict=False)
```
[SOURCE]

Function that computes the dot product between a vector `v` and the Jacobian of the given function at the point given by the inputs.

https://pytorch.org/docs/stable/autograd.html

Implementation trick—JVP

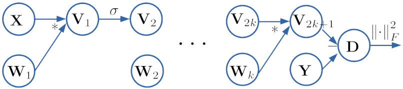
Interested in $\boldsymbol{J}_f(\boldsymbol{x})$ for $f : \mathbb{R}^n \mapsto \mathbb{R}^m$. Implement $\boldsymbol{J}_f(\boldsymbol{x})\,\boldsymbol{p}$ for any $\boldsymbol{p} \in \mathbb{R}^n$

- Why?
  * set $\boldsymbol{p} = e_i$ for $i = 1, \dots, n$ to recover columns of $\boldsymbol{J}_f(\boldsymbol{x})$
  * special structures in $\boldsymbol{J}_f(\boldsymbol{x})$ (e.g., sparsity) can be exploited
  * often enough for application
- Why possible?
  * (1) initialize partial derivatives for the input nodes as $D_{\boldsymbol{p}} v_{n-1} = p_1$, $\dots$, $D_{\boldsymbol{p}} v_0 = p_n$. (2) apply chain rule:

$$\nabla_{\boldsymbol{x}} v_i = \sum_{j:\text{incoming}} \frac{\partial v_i}{\partial v_j} \nabla_{\boldsymbol{x}} v_j \Longrightarrow D_{\boldsymbol{p}} v_i = \sum_{j:\text{incoming}} \frac{\partial v_i}{\partial v_j} D_{\boldsymbol{p}} v_j$$

  * implemented in forward-mode auto diff

Putting tricks together



Basis of implementation for: Tensorflow, Pytorch, Jax, etc
https://pytorch.org/docs/stable/autograd.html

Jax: https://github.com/google/jax http://videolectures.net/deeplearning2017_johnson_automatic_differentiation/

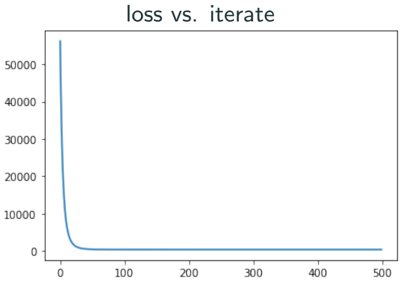
Good to know:

- In practice, graphs are built automatically by software
- Higher-order derivatives can also be done, particularly Hessian-vector product $\nabla^2 f(\boldsymbol{x})\, \boldsymbol{v}$ (Check out Jax!)
- Auto-diff in Tensorflow and Pytorch are specialized to DNNs , whereas Jax (in Python) is full fledged and more general
- General resources for autodiff: http://www.autodiff.org/, [Griewank and Walther, 2008]

Autodiff in Pytorch

Solve least squares $f(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2$ with $\nabla f(\boldsymbol{x}) = -\boldsymbol{A}^\intercal(\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x})$

```
import torch
import matplotlib.pyplot as plt

dtype = torch.float
device = torch.device("cpu")

n, p = 500, 100

A = torch.randn(n, p, device=device, dtype=dtype)
y = torch.randn(n, device=device, dtype=dtype)

x = torch.randn(p, device=device, dtype=dtype, requires_grad=True)

step_size = 1e-4

num_step = 500
loss_vec = torch.zeros(500, device=device, dtype=dtype)

for t in range(500):
  pred = torch.matmul(A, x)
  loss = torch.pow(torch.norm(y - pred), 2)

  loss_vec[t] = loss.item()

  # one line for computing the gradient
  loss.backward()

  # updates
  with torch.no_grad():
    x -= step_size*x.grad

    # zero the gradient after updating
    x.grad.zero_()

plt.plot(loss_vec.numpy())
```



Autodiff in Pytorch

Train a shallow neural network

$$f\left(\boldsymbol{W}\right) = \sum_i \left\|\boldsymbol{y}_i - \boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right\|_2^2$$

where $\sigma(z) = \max\left(z, 0\right)$, i.e., ReLU

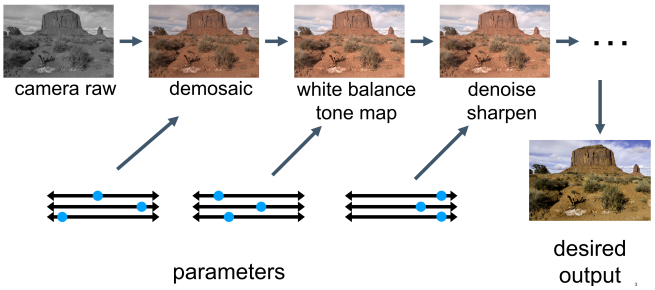
https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

- `torch.mm`
- `torch.clamp`
- `torch.no_grad()`

Back propagation is reverse mode auto-differentiation!

Outline

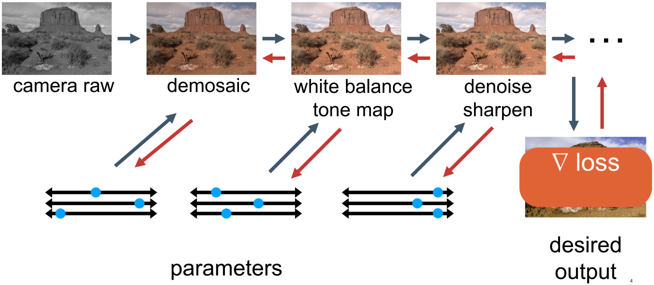
Example: image enhancement



- Each stage applies a parameterized function to the image, i.e., $q_{\boldsymbol{w}_k} \circ \cdots \circ h_{\boldsymbol{w}_3} \circ g_{\boldsymbol{w}_2} \circ f_{\boldsymbol{w}_1}(\boldsymbol{X})$ ($\boldsymbol{X}$ is the camera raw)
- The parameterized functions may or may not be DNNs
- Each function may be analytic, or simply a chunk of codes dependent on the parameters
- $\boldsymbol{w}_i$'s are the trainable parameters

Credit: https://people.csail.mit.edu/tzumao/gradient_halide/

Example: image enhancement

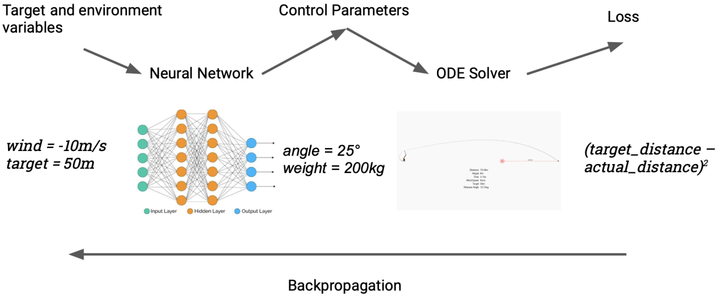


– the trainable parameters are learned by gradient descent based on auto-differentiation

– This is generalization of training DNNs with the classic feedforward structure to training general parameterized functions, using derivative-based methods

Credit: https://people.csail.mit.edu/tzumao/gradient_halide/

Example: control a trebuchet



https://fluxml.ai/blogposts/2019-03-05-dp-vs-rl/

- Given wind speed and target distance, the DNN predicts the **angle of release** and **mass of counterweight**
- Given the angle of release and mass of counterweight as initial conditions, the ODE solver calculates the actual distance by iterative methods
- We perform auto-differentiation through the iterative ODE solver and the DNN

Differential programming

Interesting resources

- Differential programming workshop @ NeurIPS'21 https://diffprogramming.mit.edu/
- Jax ecosystem https://jax.readthedocs.io/en/latest/notebooks/quickstart.html
- Notable implementations: Swift for Tensorflow https://www.tensorflow.org/swift, and Zygote in Julia https://github.com/FluxML/Zygote.jl
- Flux: machine learning package based on Zygote https://fluxml.ai/
- Taichi: differentiable programming language tailored to 3D computer graphics https://github.com/taichi-dev/taichi

Outline

Suggested reading

Autodiff in DNNs

- http://neuralnetworksanddeeplearning.com/chap2.html
- https://colah.github.io/posts/2015-08-Backprop/
- http://videolectures.net/deeplearning2017_johnson_automatic_differentiation/

Yes you should understand backprop

- https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b

Differentiable programming

- https://en.wikipedia.org/wiki/Differentiable_programming
- https://fluxml.ai/2019/02/07/what-is-differentiable-programming.html
- https://fluxml.ai/2019/03/05/dp-vs-rl.html

[Baydin et al., 2017] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2017). **Automatic differentiation in machine learning: a survey.** *The Journal of Machine Learning Research*, 18(1):5595–5637.

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