

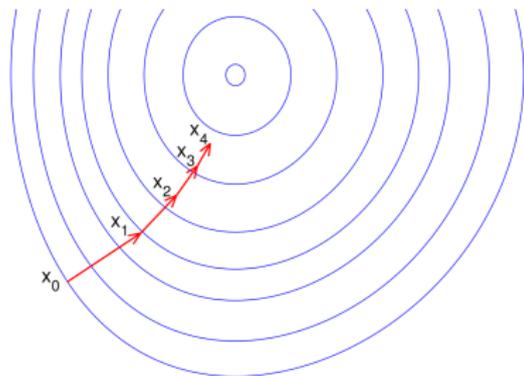
Basics of Numerical Optimization: Computing Derivatives

Ju Sun

Computer Science & Engineering
University of Minnesota, Twin Cities

February 12, 2026

Derivatives for numerical optimization

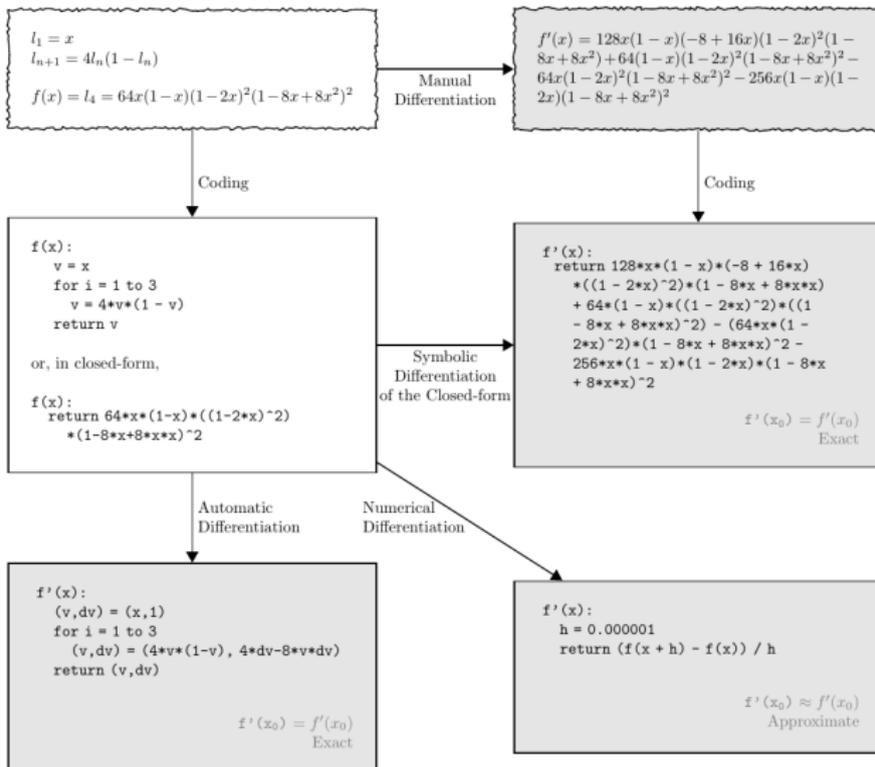


Credit: aria42.com

- 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 $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$
 - * 2nd order methods: use $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$ and $\nabla^2 f(\mathbf{x})$
 - **Numerical (not analytical) derivatives** (i.e., numbers) needed for the iterations

This lecture: how to compute the numerical derivatives

Four kinds of computing techniques



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 $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $h : \mathbb{R}^n \rightarrow \mathbb{R}^k$, and f is differentiable at \mathbf{x} and $z = h(\mathbf{y})$ is differentiable at $\mathbf{y} = f(\mathbf{x})$. Then, $z = h \circ f(\mathbf{x}) : \mathbb{R}^m \rightarrow \mathbb{R}^k$ is differentiable at \mathbf{x} , and

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

When $k = 1$,

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

– **Perturbation-expansion method**

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

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

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

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`)
- Mathematica (`D`)
- Matrix Calculus <https://www.matrixcalculus.org/>

Effective for simple functions

Analytical differentiation

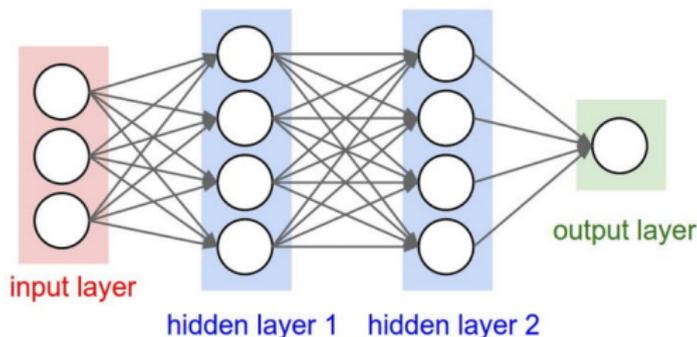
Finite-difference approximation

Automatic differentiation

Differentiable programming

Suggested reading

Limitation of analytical differentiation



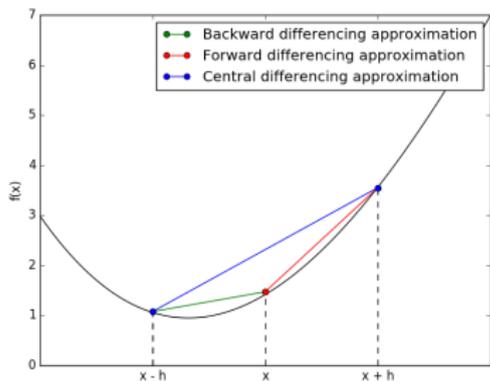
What is the gradient and/or Hessian of

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

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

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

Approximate the gradient



(Credit: numex-blog.com)

$$f'(x) = \lim_{\delta \rightarrow 0} \frac{f(x+\delta) - f(x)}{\delta} \approx \frac{f(x+\delta) - f(x)}{\delta}$$

with δ sufficiently small

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

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

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

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

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

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

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

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

central themes can also be derived

Stronger forms of Taylor's theorems

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

Why the central theme is better?

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

Approximate the Hessian

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

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

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

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

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

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

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

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)

Analytical differentiation

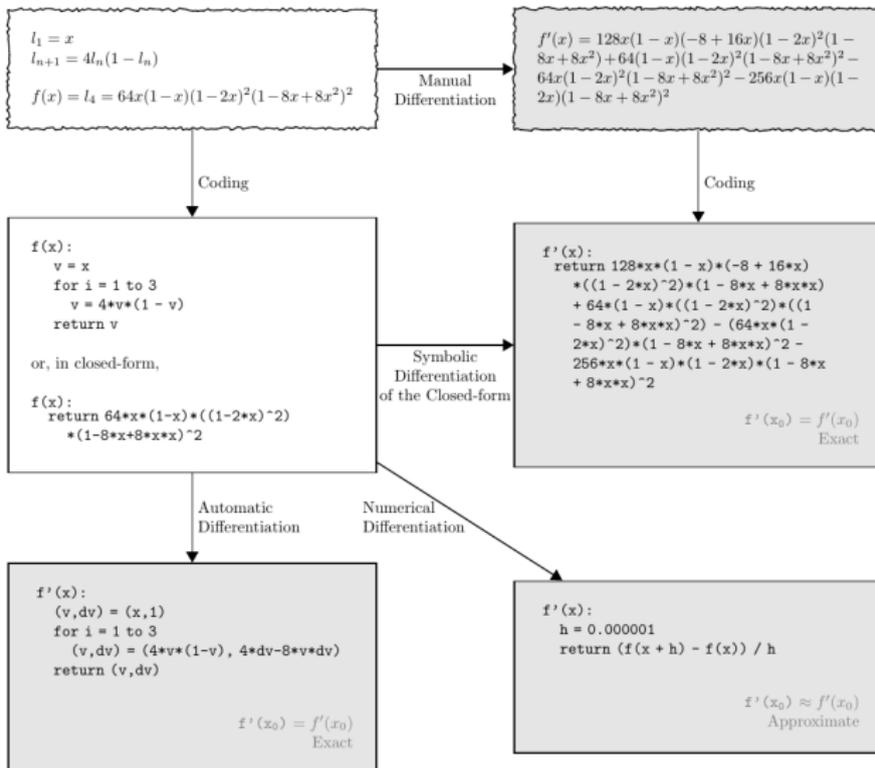
Finite-difference approximation

Automatic differentiation

Differentiable programming

Suggested reading

Four kinds of computing techniques



Auto differentiation (auto diff, AD) in 1D

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



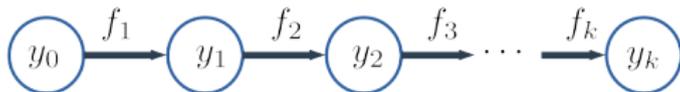
Chain rule in Leibniz form:

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

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

Example: For $f(x) = (x^2 + 1)^2$, 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: $\frac{df}{dx} = \frac{df}{dy_0} = \left(\left(\left(\left(\left(\left(\frac{dy_k}{dy_{k-1}} \right) \frac{dy_{k-1}}{dy_{k-2}} \right) \dots \right) \frac{dy_2}{dy_1} \right) \frac{dy_1}{dy_0} \right) \right)$

Example: For $f(x) = (x^2 + 1)^2$, 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**:

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

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

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

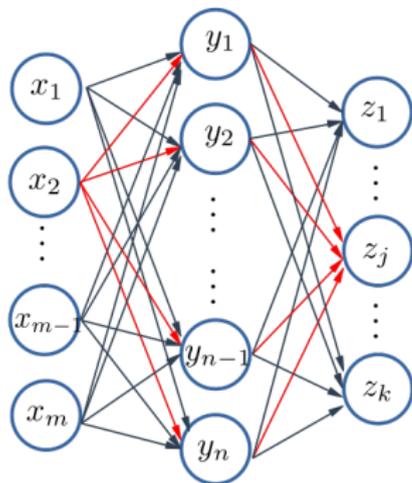
i.e., starting from the leaf: $\frac{dy_k}{dy_k} \mapsto \frac{dy_k}{dy_{k-1}} \mapsto \frac{dy_k}{dy_{k-2}} \mapsto \dots \mapsto \frac{dy_k}{dy_0}$

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

Auto differentiation in high dimensions

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

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

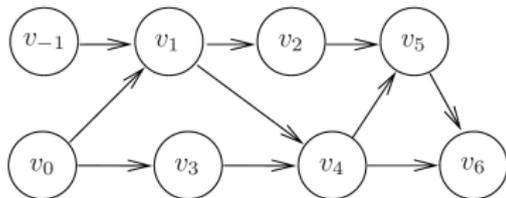


- Each node is a variable, as a function of all incoming variables
- If node B a child of node A , $\frac{\partial B}{\partial A}$ is **the rate of change in B wrt change in A**
- Traveling along a path, rates of changes should be **multiplied**
- Chain rule: **summing up rates over all connecting paths!**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

NB: this is a computational graph, not a NN

A multivariate example—forward mode

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



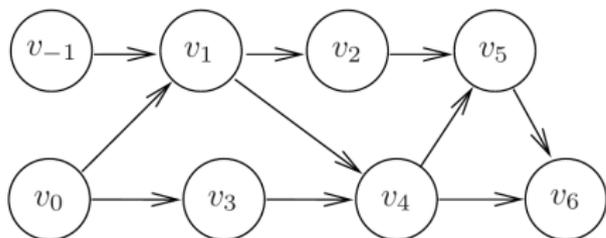
v_{-1}	$= x_1$	$= 1.5000$
v_0	$= x_2$	$= 0.5000$
v_1	$= v_{-1}/v_0$	$= 1.5000/0.5000 = 3.0000$
v_2	$= \sin(v_1)$	$= \sin(3.0000) = 0.1411$
v_3	$= \exp(v_0)$	$= \exp(0.5000) = 1.6487$
v_4	$= v_1 - v_3$	$= 3.0000 - 1.6487 = 1.3513$
v_5	$= v_2 + v_4$	$= 0.1411 + 1.3513 = 1.4924$
v_6	$= v_5 * v_4$	$= 1.4924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

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

- 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:
 $O(\#edges + \#nodes)$
- for $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, make n forward passes: $O(n(\#edges + \#nodes))$

A multivariate example—reverse mode



```
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_bar_6 = y = 1.0000
v_bar_5 = v_bar_6 * v_4 = 1.0000 * 1.3513 = 1.3513
v_bar_4 = v_bar_5 * v_5 = 1.0000 * 1.4924 = 1.4924
v_bar_4 = v_bar_4 + v_bar_5 = 1.4924 + 1.3513 = 2.8437
v_bar_2 = v_bar_5 = 1.3513
v_bar_3 = -v_bar_4 = -2.8437
v_bar_1 = v_bar_4 = 2.8437
v_bar_0 = v_bar_3 * v_3 = -2.8437 * 1.6487 = -4.6884
v_bar_1 = v_bar_1 + v_bar_2 * cos(v_1) = 2.8437 + 1.3513 * (-0.9900) = 1.5059
v_bar_0 = v_bar_0 - v_bar_1 * v_1/v_0 = -4.6884 - 1.5059 * 3.000/0.5000 = -13.7239
v_bar_{-1} = v_bar_1/v_0 = 1.5059/0.5000 = 3.0118
x_bar_2 = v_bar_0 = -13.7239
x_bar_1 = v_bar_{-1} = 3.0118
```

– interested in $\frac{\partial y}{\partial}$; for each variable v_i , write $\bar{v}_i \doteq \frac{\partial y}{\partial v_i}$ (called **adjoint variable**)

– for each node, sum up partials over all outgoing edges, e.g.,

$$\bar{v}_4 = \frac{\partial v_5}{\partial v_4} \bar{v}_5 + \frac{\partial v_6}{\partial v_4} \bar{v}_6$$

– complexity:

$$O(\#edges + \#nodes)$$

– for $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$, make m backward passes:

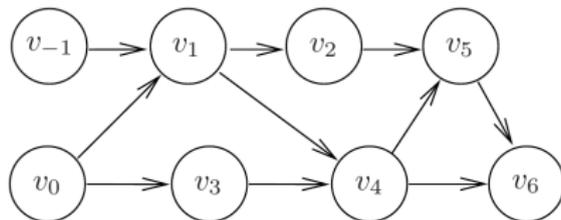
$$O(m(\#edges + \#nodes))$$

example from Ch 1

of [Griewank and Walther, 2008]

Forward vs. reverse modes

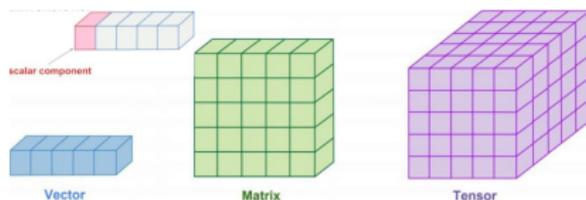
For general function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, suppose there is no loop in the computational graph, i.e., **acyclic graph**. E : set of edges ; V : set of nodes



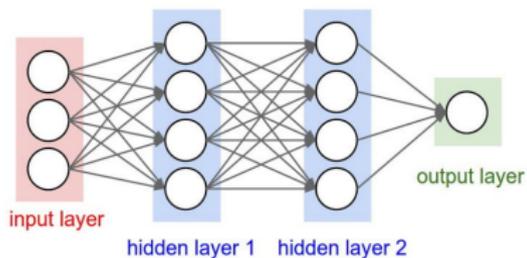
	forward mode	reverse mode
start from	roots	leaves
end with	leaves	roots
invariants	$\dot{v}_i \doteq \frac{\partial v_i}{\partial x}$ (x —root of interest)	$\bar{v}_i \doteq \frac{\partial y}{\partial v_i}$ (y —leaf of interest)
rule	sum over incoming edges	sum over outgoing edges
computation	$O(n E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

Implementation trick—tensor abstraction

Tensors: multi-dimensional arrays

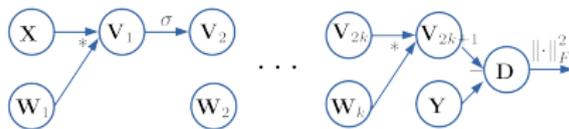


Each node in the computational graph can be a tensor (scalar, vector, matrix, 3-D tensor, ...)

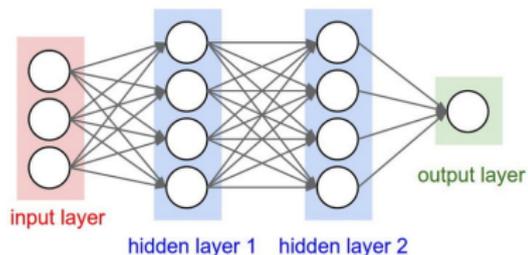


$$f(\mathbf{W}) = \|\mathbf{Y} - \sigma(\mathbf{W}_k \sigma(\mathbf{W}_{k-1} \sigma \dots (\mathbf{W}_1 \mathbf{X})))\|_F^2$$

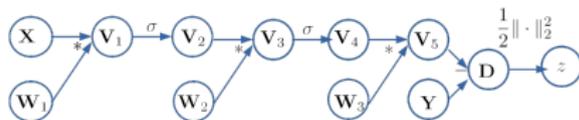
computational graph for DNN



Implementation trick—tensor abstraction



computational graph for DNN



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

- neater computational graph
- tensor (i.e., vector) chain rules apply, often in tensor-free computation

Fact: For two matrices (tensors) D and M of compatible size, where D is fixed and M is a function of M'

$$\nabla_{M'} \langle M, D \rangle = \mathcal{J}_{M' \rightarrow M}^T(M') [D]$$

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

Implementation trick—VJP

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

- Why?
 - * set $\mathbf{v} = \mathbf{e}_i$ for $i = 1, \dots, m$ to recover rows of $\mathbf{J}_f(\mathbf{x})$
 - * special structures in $\mathbf{J}_f(\mathbf{x})$ (e.g., sparsity) can be exploited
 - * often enough for application, e.g., calculate $\nabla(g \circ f) = (\nabla f^\top \mathbf{J}_f)^\top$ with known ∇f
- Why possible?
 - * $\mathbf{v}^\top \mathbf{J}_f(\mathbf{x}) = \mathbf{J}_{\mathbf{v}^\top f}(\mathbf{x})$ so keep track of $\frac{\partial}{\partial v_i}(\mathbf{v}^\top f) = \sum_{k:\text{outgoing}} \frac{\partial v_k}{\partial v_i} \frac{\partial}{\partial v_k}(\mathbf{v}^\top f)$
 - * implemented 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 $\mathbf{J}_f(\mathbf{x})$ for $f: \mathbb{R}^n \mapsto \mathbb{R}^m$. Implement $\mathbf{J}_f(\mathbf{x})\mathbf{p}$ for any $\mathbf{p} \in \mathbb{R}^n$

– Why?

- * set $\mathbf{p} = \mathbf{e}_i$ for $i = 1, \dots, n$ to recover columns of $\mathbf{J}_f(\mathbf{x})$
- * special structures in $\mathbf{J}_f(\mathbf{x})$ (e.g., sparsity) can be exploited
- * often enough for application

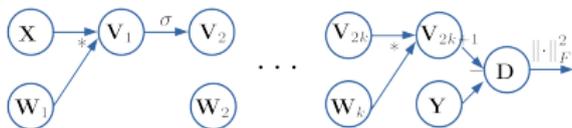
– Why possible?

- * (1) initialize partial derivatives for the input nodes as $D_{\mathbf{p}}v_{n-1} = p_1, \dots, D_{\mathbf{p}}v_0 = p_n$. (2) apply chain rule:

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

- * 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://videlectures.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 $\nabla^2 f(x) v$ (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 $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2$ with $\nabla f(\mathbf{x}) = -\mathbf{A}^T(\mathbf{y} - \mathbf{A}\mathbf{x})$

```
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()

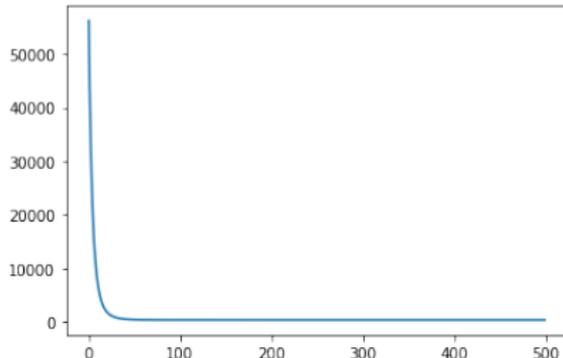
    # one line for computing the gradient
    loss.backward()

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

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

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

loss vs. iterate



Autodiff in Pytorch

Train a shallow neural network

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

where $\sigma(z) = \max(z, 0)$, 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!

Analytical differentiation

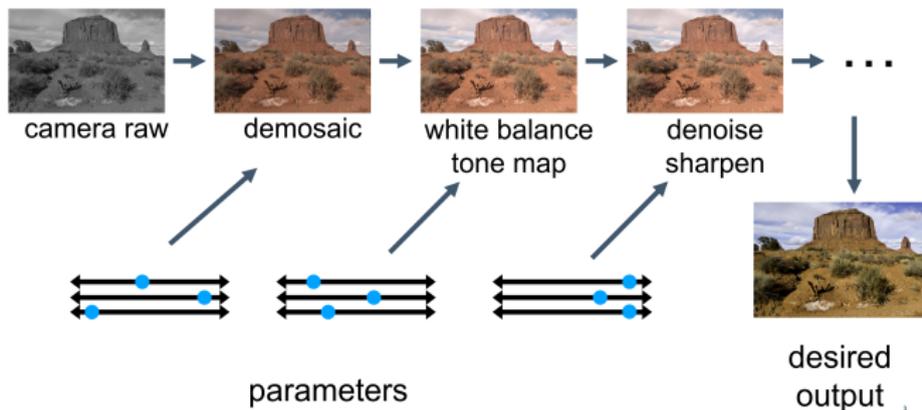
Finite-difference approximation

Automatic differentiation

Differentiable programming

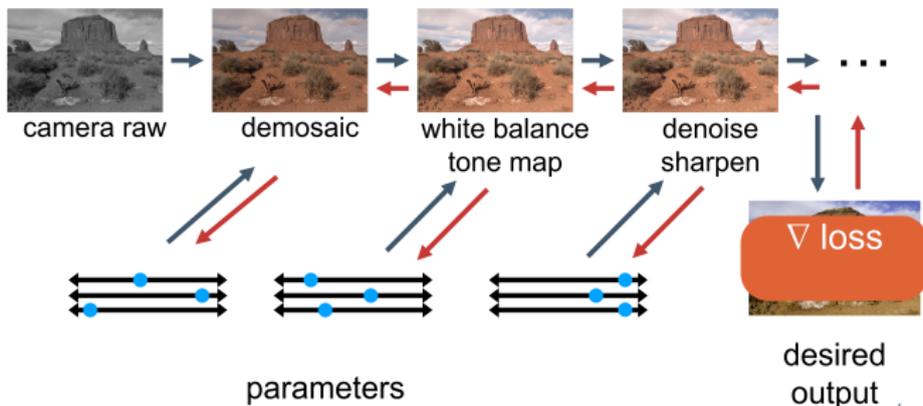
Suggested reading

Example: image enhancement



- Each stage applies a parameterized function to the image, i.e., $q_{w_k} \circ \dots \circ h_{w_3} \circ g_{w_2} \circ f_{w_1}(\mathbf{X})$ (\mathbf{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
- w_i 's are the trainable parameters

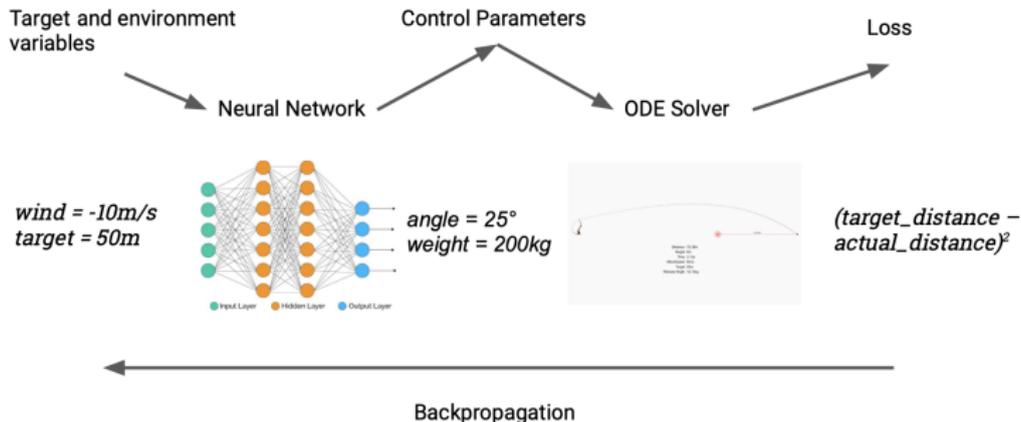
Example: image enhancement



- 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/tzumaogradient_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>

Analytical differentiation

Finite-difference approximation

Automatic differentiation

Differentiable programming

Suggested reading

Suggested reading

Autodiff in DNNs

- <http://neuralnetworksanddeeplearning.com/chap2.html>
- <https://colah.github.io/posts/2015-08-Backprop/>
- http://videlectures.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.