Automatic Differentiation

An overview of three frameworks for differentiation

Numerical Differentiation:

$$\frac{df}{dx} = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

$$\frac{\partial^2 f}{\partial x \partial y} \approx \frac{f(x+h, y+k) - f(x+h, y-k) - f(x-h, y+k) + f(x-h, y-k)}{4hk}$$

Pros: Absurdly easy to implement. Taught in school.

Cons: Prone to round-off / truncation errors. For every input dimension at least two evaluations of the function is necessary.

Symbolic Differentiation:

$$e^x \xrightarrow{d/dx} e^x \qquad x^2 y^3 \xrightarrow{\frac{\partial^2}{\partial x \partial y}} 6xy^2$$

Pros: You get exact derivatives. Students start learning about differentiation by doing symbolic differentiation.

Cons: May suffer from "expression swelling" which refers to the phenomenon where the expression for derivative is significantly more complicated or difficult to evaluate than the original function. Although this has been disputed and blamed on implementation by Sören Laue[1] (arxiv 2020).

Automatic Differentiation :

AD tries to combine the best of both worlds. You don't get a symbolic representation of the derivative. But you do get exact numerical values for the derivative up to machine precision.

Pros : Compared to numerical differentiation AD is a lot faster and more precise, especially for higher order derivatives. In fact, numerical differentiation is so slow that before AD became widely used in ML libraries, programmers would often symbolically differentiate loss functions by hand.

[2]

Method / Runtime	LV (μs)	BRUSS (s)	POLLU (s)	PKPD (ms)
DSA AD	120	0.65	0.015	1.64
Numerical diff	670	3.9	0.036	12.3

LV — Non-stiff Lotka-Volterra equations

BRUSS — 2D stiff Brusselator reaction-diffusion PDE

POLLU — A stiff pollution model

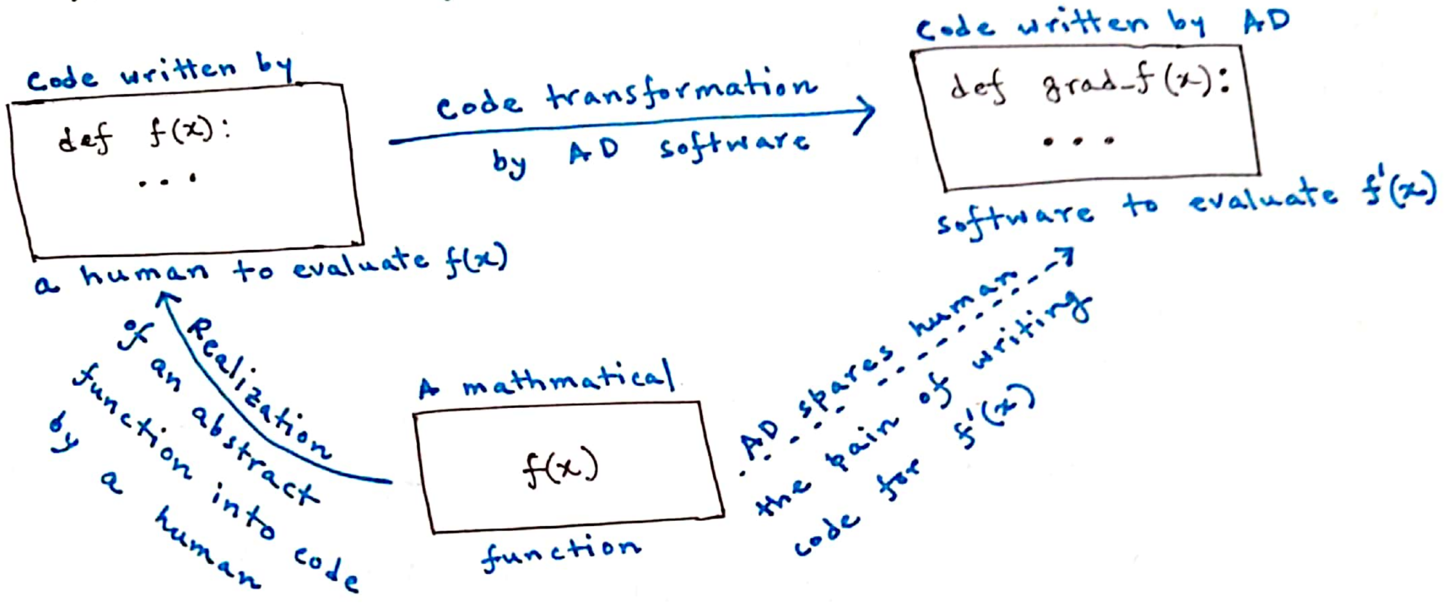
PKPD — A non-stiff pharmacokinetic/pharmacodynamic system

solved using both a modified version of AD and numerical differentiation.

Cons : Implementation is nuanced. I wasn't taught this at school.

Forward Mode Automatic Differentiation

How AD actually works:



Wengert lists:

Suppose your favorite programming languge has a module/ library of primitive functions that either take 1D or 2D inputs (e.g. tensorflow. math). Suppose this module, let's call it G, looks like this $G = \{ \underset{2D}{\text{add}}, \underset{2D}{\text{subtract}}, \underset{2D}{\text{multiply}}, \underset{2D}{\text{divide}}, \underset{1D}{\text{sin}}, \underset{1D}{\text{cos}}, \underset{1D}{\text{tan}}, \underset{1D}{\text{square}}, \underset{2D}{\text{pow}}, \underset{1D}{\text{exp}}, \underset{1D}{\text{log}}, \ldots \}$.

The members of G are so elementary that their derivatives are known in analytical form and stored in another module G'.

A Wengert list is simply a collection of assignment statements that construct a complicated function from the primitive or simple functions in G.

Examples :

$f(x_1, x_2) = x_1 x_2 + \sin x_1$

Wengert list / code for f

$w_1 = x_1$

$w_2 = x_2$

$w_3 = w_1 w_2$

$w_4 = \sin w_1$

$w_5 = w_3 + w_4$

$g(y, \mu, \sigma) = -\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2 - \log \sigma$

Wengert list / code for g

$w_1 = y$

$w_2 = \mu$

$w_3 = \sigma$

$w_4 = w_1 - w_2$

$w_5 = w_4 / w_3$

$w_6 = w_5^2$

$w_7 = -\frac{1}{2} w_6$

$w_8 = \log w_3$

$w_9 = w_7 - w_8$

Once you have a Wengert list or code for a function you can easily create a dual Wengert list to compute the derivative since each assignment in the list uses only an elementary function whose derivative is already known.

Dual Wengert list for $\frac{\partial f}{\partial x_2}$ $\left(\cdot = \frac{\partial}{\partial x_2}\right)$

$\dot{w}_1 = 0$

$\dot{w}_2 = 1$

$\dot{w}_3 = \dot{w}_1 w_2 + w_1 \dot{w}_2 = 0 \cdot x_2 + x_1 \cdot 1$
$= x_1$

$\dot{w}_4 = \cos(w_1) \cdot \dot{w}_1 = \cos(x_1) \cdot 0 = 0$

$\dot{w}_5 = \dot{w}_3 + \dot{w}_4 = x_1 + 0 = x_1$

Dual Wengert list for $\frac{\partial g}{\partial \mu}$ $\left(\cdot = \frac{\partial}{\partial \mu}\right)$

$\dot{w}_1 = 0$

$\dot{w}_2 = 1$

$\dot{w}_3 = 0$

$\dot{w}_4 = -1$

$\dot{w}_5 = \frac{-\sigma - 0}{\sigma^2} = -\frac{1}{\sigma}$

$\dot{w}_6 = 2 w_5 \dot{w}_5 = \frac{-2(y-\mu)}{\sigma^2}$

$\dot{w}_7 = \frac{y-\mu}{\sigma^2}$

$\dot{w}_8 = \frac{\dot{w}_3}{w_3} = 0$

$\dot{w}_9 = \dot{w}_7 = \frac{y-\mu}{\sigma^2}$

Note that in the dual lists x_1, x_2, y, μ, σ are not symbols but numeric quantities e.g. to compute $\frac{\partial g}{\partial \mu}$ at $(y, \mu, \sigma) = (1, 0, 1)$ at each step the RHS would be a numeric quantity (and therefore "expression swelling" would be impossible) :

Dual list for $\frac{\partial g}{\partial \mu}$ at $(y, \mu, \sigma) = (1, 0, 1)$

$\dot{w}_1 = 0$

$\dot{w}_2 = 1$

$\dot{w}_3 = 0$

$\dot{w}_4 = -1$

$\dot{w}_5 = -1$

$\dot{w}_6 = -2$

$\dot{w}_7 = 1$

$\dot{w}_8 = 0$

$\dot{w}_9 = 1$

Wengert list is named after the person whose PhD thesis in 1974 laid out the foundations of automatic differentiation

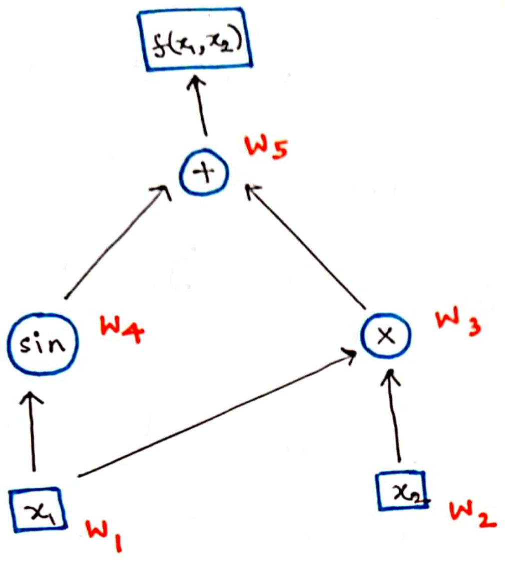
Computational Graph / Expression Graph :

Computational graphs are directed acyclic graphical representation of Wengert lists. Many modern ML libraries (e.g. tensorflow) are built using the concept of a computational graph. Every node in the graph is called an op (operation) in tensorflow which makes perfect sense after looking at a few examples.

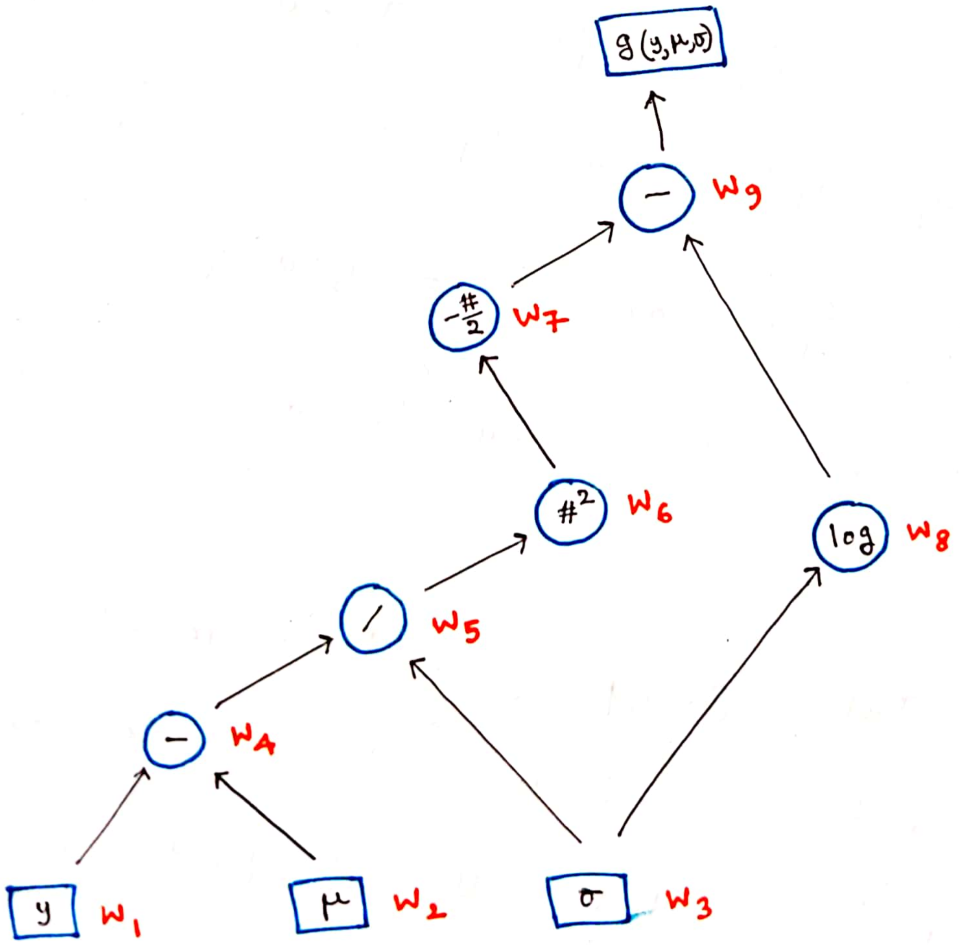
Scanned with CamScanner

Examples :

$$f(x_1, x_2) = x_1 x_2 + \sin x_1$$



$$g(y, \mu, \sigma) = -\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2 - \log \sigma$$



Forward AD on a computation graph:

Suppose we want to compute the Jacobian matrix J of a target function $f: \mathbb{R}^n \to \mathbb{R}^m$ i.e. $J_{ij} = \frac{\partial f_i}{\partial x_j}$. If f is a composite function (which Wengert lists are) and $f = f^L \circ f^{L-1} \circ \dots \circ f^1$ then corresponding Jacobian satisfies $J = J_L \cdot J_{L-1} \cdots J_1$ by chain-rule.

Let $u \in \mathbb{R}^n$. One application / sweep of forward-mode AD numerically evaluates the action of the Jacobian matrix J on u i.e. $J \cdot u$.

$$J \cdot u = J_L \cdot J_{L-1} \cdots J_1 \cdot u$$

$$= J_L \cdot J_{L-1} \cdots J_2 \cdot u_1 = \dots = J_L \cdot u_{L-1}$$

where $u_1 = J_1 \cdot u$ and $u_\ell = J_\ell \cdot u_{\ell-1}$

Moreover, it computes $J \cdot u$ in a matrix-free fashion.

Example: Take $g(y, \mu, \sigma)$. Here $m = 1$ and $J = \nabla g$ and so $J \cdot u$ = partial derivative of g along u. Let $u = (\alpha, \beta, \gamma)$.

The dual list for $\nabla g \cdot u$

$$\dot{w}_1 = \alpha$$

$$\dot{w}_2 = \beta$$

$$\dot{w}_3 = \gamma$$

$$\dot{w}_4 = \dot{w}_1 - \dot{w}_2 = \alpha - \beta$$

$$\dot{w}_5 = \dot{\left(\frac{w_4}{w_3}\right)} = \frac{w_3 \dot{w}_4 - \dot{w}_3 w_4}{w_3^2} = \frac{\sigma(\alpha-\beta) - \gamma(y-\mu)}{\sigma^2}$$

$$\dot{w}_6 = 2 w_5 \dot{w}_5 = 2 \cdot \frac{y-\mu}{\sigma} \cdot \frac{\sigma(\alpha-\beta) - \gamma(y-\mu)}{\sigma^2}$$

$$\dot{w}_7 = \frac{y-\mu}{\sigma^2}\left[\frac{\gamma(y-\mu)}{\sigma} - (\alpha-\beta)\right]$$

$$\dot{w}_8 = \frac{\dot{w}_3}{w_3} = \frac{\gamma}{\sigma}$$

$$\dot{w}_9 = \dot{w}_7 - \dot{w}_8 = \frac{y-\mu}{\underline{\sigma^2}}\left[\frac{\gamma(y-\mu)}{\underline{\sigma}} - \underline{(\alpha-\beta)}\right] - \frac{\gamma}{\underline{\sigma}}$$

which is exactly what we expect by direct computation

$$\nabla g \cdot u = \left[\underline{-\frac{y-\mu}{\sigma^2}} \quad \underline{\frac{y-\mu}{\sigma^2}} \quad \underline{\frac{(y-\mu)^2}{\sigma^3} - \frac{1}{\sigma}} \right] \begin{bmatrix} \underline{\alpha} \\ \underline{\beta} \\ \underline{\gamma} \end{bmatrix}$$

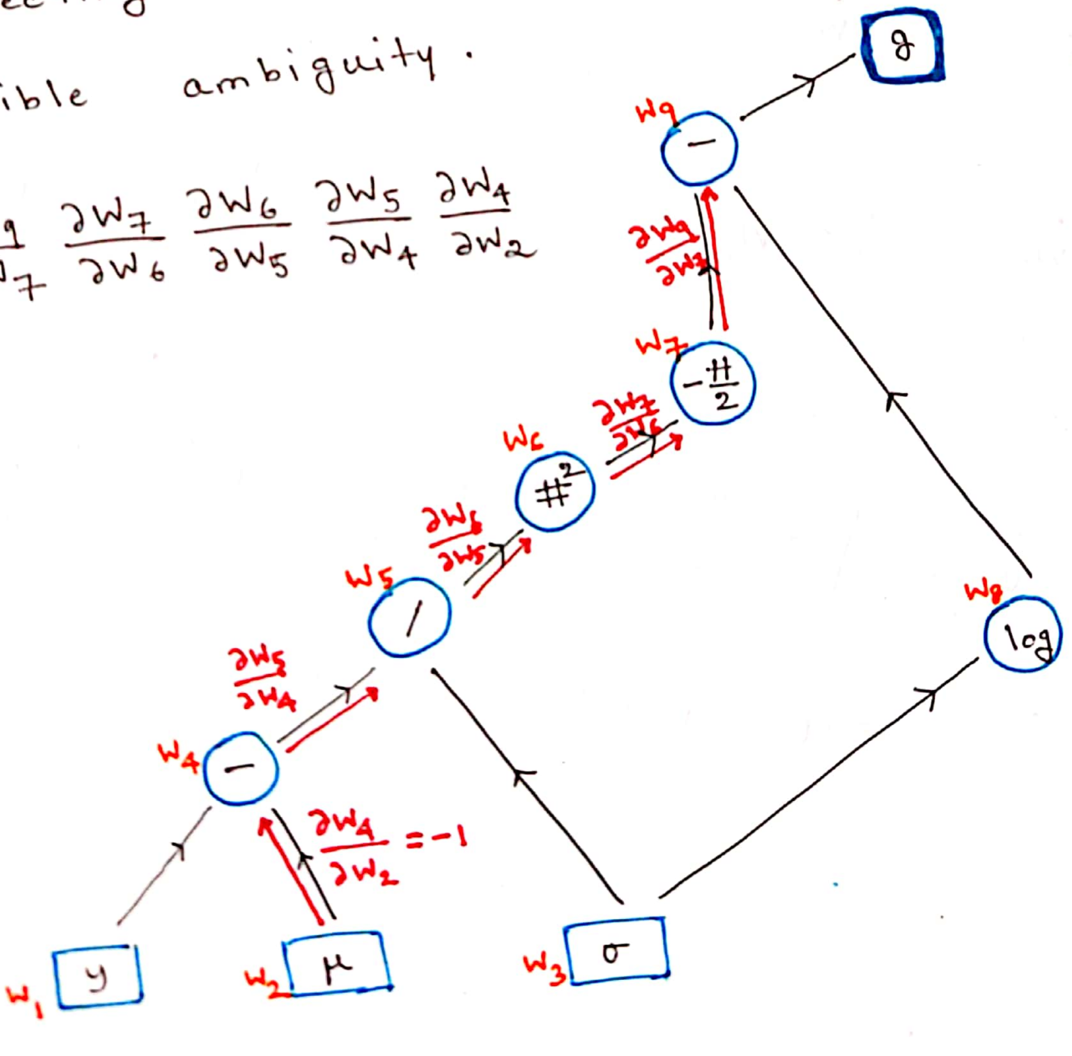
$$= -\frac{(y-\mu)}{\sigma^2}(\alpha-\beta) + \frac{\gamma(y-\mu)^2}{\sigma^3} - \frac{\gamma}{\sigma}$$

A forward sweep on the computation graph:

A forward sweep on the computation graph computes a derivative for each edge and accumulates them accordingly to produce the same result as the dual list.

For example to compute $\frac{\partial g}{\partial \mu}$ using a forward sweep you only need to compute derivatives along the path connecting g and μ and combine them using chain rule. Note that in a directed acyclic graph there would a single directed path connecting [redacted] g and μ which rules out any possible ambiguity.

$$\frac{\partial w_9}{\partial w_2} = \frac{\partial w_9}{\partial w_7}\frac{\partial w_7}{\partial w_6}\frac{\partial w_6}{\partial w_5}\frac{\partial w_5}{\partial w_4}\frac{\partial w_4}{\partial w_2}$$



g
w_9
$-$
$\frac{\partial w_9}{\partial w_7}$
w_7
$-\frac{\#}{2}$
$\frac{\partial w_7}{\partial w_6}$
w_6
$\#^2$
$\frac{\partial w_6}{\partial w_5}$
w_5
$/$
$\frac{\partial w_5}{\partial w_4}$
w_4
$-$
$\frac{\partial w_4}{\partial w_2} = -1$
w_8
log
w_1 y
w_2 μ
w_3 σ

All the derivatives appearing in the chain of $\frac{\partial w_9}{\partial w_2}$ are analytically available to us because each node is an elementary operation.

$$\frac{\partial w_4}{\partial w_2} = -1 \, , \quad \frac{\partial w_5}{\partial w_4} = \frac{1}{\sigma} \, , \quad \frac{\partial w_6}{\partial w_5} = 2w_5 = 2\frac{w_4}{w_3} = 2\frac{y-\mu}{\sigma} ,$$

$$\frac{\partial w_7}{\partial w_6} = -\frac{1}{2} \, , \quad \frac{\partial w_9}{\partial w_7} = 1 \quad \Rightarrow \quad \frac{\partial w_9}{\partial w_2} = \frac{y-\mu}{\sigma^2}$$

$$\therefore \quad \frac{\partial g}{\partial \mu} = \frac{y-\mu}{\sigma^2}$$

Complexity of Forward AD:

For an arbitrary function choosing the right u will yield a column of J. So to compute the full Jacobian matrix of size $m \times n$ we would need n forward sweeps. Using fused-multiply adds, denoted OPS, as a metric for computational complexity, [3]

$$\text{OPS}(f(\bar{x}), J.u) \leq 2.5 \ \text{OPS}(f(\bar{x}))$$

In the LHS both $f(\bar{x})$ and $J.u$ appear together because during a forward sweep both the function and its derivative are computed together (simply evaluating the nodes gives you $f(\bar{x})$ which is required to compute $J.u$).

The Math behind Forward AD

Dual Numbers:

Extend all numbers by adding a second component

$$x \mapsto x + \dot{x}d$$

- d is just a symbol distinguishing the second component
- d is analogous to $i = \sqrt{-1}$
- But $d^2 = 0$, as opposed to $i^2 = -1$

Arithmatic on dual numbers:

$$(x + \dot{x}d) + (y + \dot{y}d) = x + y + (\dot{x} + \dot{y})d$$

$$(x + \dot{x}d) \cdot (y + \dot{y}d) = xy + (x\dot{y} + \dot{x}y)d$$

$$-(x + \dot{x}d) = -x - \dot{x}d$$

$$\frac{1}{x + \dot{x}d} \quad (x \neq 0) = \frac{x - \dot{x}d}{(x + \dot{x}d)(x - \dot{x}d)} = \frac{x - \dot{x}d}{x^2}$$

$$= \frac{1}{x} - \frac{\dot{x}}{x^2} d$$

If $P(x) = p_0 + p_1 x + \dots + p_n x^n$

$$P(x + \dot{x}d) = p_0 + p_1(x + \dot{x}d) + \dots + p_n(x + \dot{x}d)^n$$

$$= p_0 + p_1 x + \dots + p_n x^n + p_1 \dot{x}d + 2p_2 x\dot{x}d + \dots + np_n x^{n-1}\dot{x}d$$

$$= P(x) + P'(x) \cdot \dot{x}d$$

The pattern is now clear and we can now either guess or derive using Taylor series,

$$\sin(x+\dot{x}d) = \sin x + \cos x \; \dot{x} d$$
$$\cos(x+\dot{x}d) = \cos x - \sin x \; \dot{x} d$$
$$e^{x+\dot{x}d} = e^{x} + e^{x} \dot{x} d$$
$$\log(x+\dot{x}d) = \log x + \frac{\dot{x}}{x} d \quad (x \neq 0)$$
$$\sqrt{x+\dot{x}d} = \sqrt{x} + \frac{\dot{x}}{2\sqrt{x}} d \quad (x \neq 0) \qquad \text{etc.}$$

Also,

$$f \circ g(x+\dot{x}d) = f(g(x) + g'(x)\cdot\dot{x}d) \quad [\because g(x+\dot{x}d) = g(x)+g'(x)\dot{x}d]$$

$$= f(g(x)) + f'(g(x))\, g'(x)\, \dot{x}d$$

It can be shown that Forward AD is equivalent to evaluating a function at a dual point (one easy to spot similarity is that both evaluate $f(x)$ and $f'(x)$ together). For higher dimensional inputs higher dimensional dual numbers can be used to justify this equivalence.

Example in 2D: $f(x_1, x_2) = x_1 x_2 + \sin x_1$

$$f(x_1+\dot{x}_1 d_1, x_2+\dot{x}_2 d_2) = (x_1+\dot{x}_1 d_1)(x_2+\dot{x}_2 d_2) + \sin(x_1+\dot{x}_1 d_1)$$

$$= x_1 x_2 + \overset{\sin x_1}{\wedge} + (x_2 + \cos(x_1))\, \dot{x}_1 d_1 + x_1 \dot{x}_2 d_2 \qquad (d_i d_j = 0)$$

$$= f(x_1, x_2) + \frac{\partial f}{\partial x_1}\dot{x}_1 d_1 + \frac{\partial f}{\partial x_2}\dot{x}_2 d_2$$

Reverse Mode Automatic Differentiation

Forward AD works bottom-up i.e. it follows the direction of the edges. But we can also apply chain rule in a top-down fashion.

Adjoints:

Adjoint of a node w_i in the computation graph is $\bar{w}_i = \frac{\partial z}{\partial w_i}$ where z is a top / output node. For example, adjoint of w_5 is $\frac{\partial w_9}{\partial w_5}$ in the graph of $g(y, \mu, \sigma)$, Reverse AD computes an adjoint for each node and combines them using chain rule. This takes place in two steps — 1) Forward evaluation trace, 2) Reverse adjoint trace.

Reverse AD in action: Suppose we want to evaluate $\frac{\partial g}{\partial \mu}$ at $(y, \mu, \sigma) = (1, 0, 1)$.

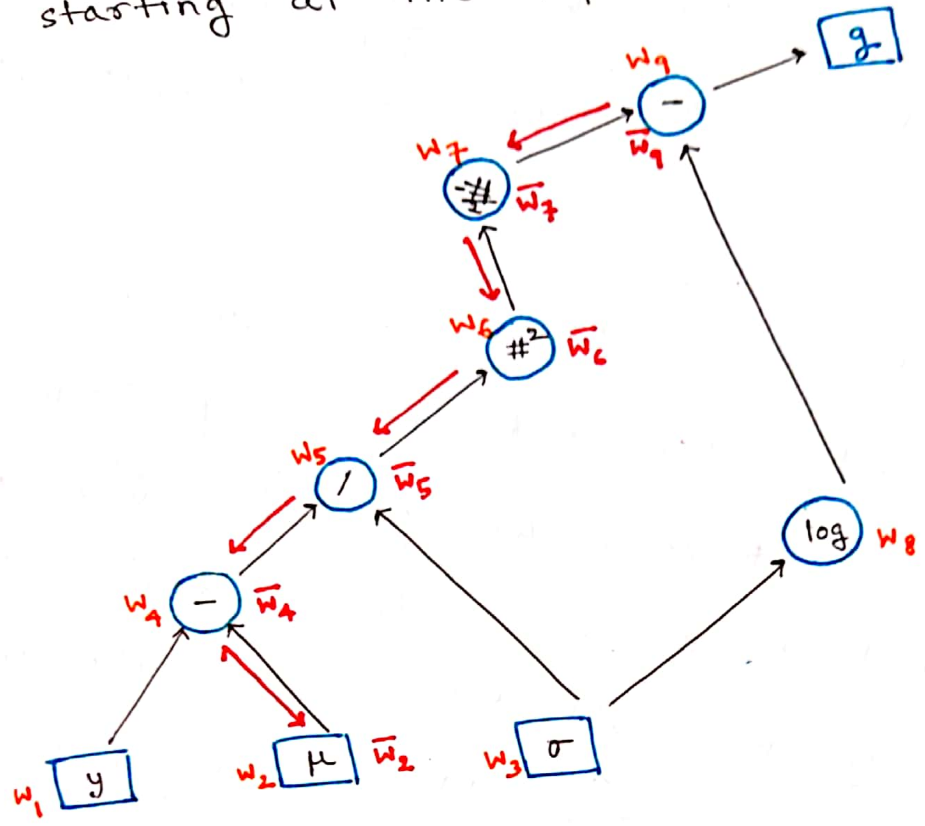
Forward Evaluation Trace: Forward trace is just evaluation of the Wengert list.

$w_1 = 1$, $w_2 = 0$, $w_3 = 1$, $w_4 = w_1 - w_2 = 1$

$w_5 = \frac{w_4}{w_3} = 1$, $w_6 = w_5^2 = 1$, $w_7 = -\frac{1}{2}w_6 = -\frac{1}{2}$,

$w_8 = \log 1 = 0$, $w_9 = w_7 - w_8 = -\frac{1}{2}$

Reverse Adjoint trace: We evaluate adjoint for each node starting at the top.



$$\bar{w}_9 = \frac{\partial w_9}{\partial w_9} = 1$$

$$\bar{w}_7 = \frac{\partial w_9}{\partial w_7} = 1$$

$$\bar{w}_6 = \frac{\partial w_9}{\partial w_6} = \frac{\partial w_9}{\partial w_7}\frac{\partial w_7}{\partial w_6} = \bar{w}_7 \frac{\partial w_7}{\partial w_6} = -\frac{1}{2}$$

$$\bar{w}_5 = \frac{\partial w_9}{\partial w_5} = \frac{\partial w_9}{\partial w_6}\frac{\partial w_6}{\partial w_5} = \bar{w}_6 \frac{\partial w_6}{\partial w_5} = -\frac{1}{2}\cdot 2 \cdot 1 = -1$$

$$\bar{w}_4 = \frac{\partial w_9}{\partial w_4} = \frac{\partial w_9}{\partial w_5}\frac{\partial w_5}{\partial w_4} = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = -1 \cdot \frac{1}{1} = -1$$

$$\bar{w}_2 = \frac{\partial w_9}{\partial w_2} = \frac{\partial w_9}{\partial w_4}\frac{\partial w_4}{\partial w_2} = \bar{w}_4 \frac{\partial w_4}{\partial w_2} = -1\cdot(-1) = 1$$

$$\therefore \frac{\partial g}{\partial \mu} = \bar{w}_2 = 1$$

Takeaway: Although w_9 and w_2 are not neighbors, in the end to compute $\bar{w}_2$ we only needed to compute $\frac{\partial w_4}{\partial w_2}$ which is easy to compute since there's an edge between w_4 and w_2.

Similarly, to compute $\overline{w_6}$ the only new thing we needed to compute was $\frac{\partial w_7}{\partial w_6}$ which is easy since w_7 is an elementary function of w_6 and so on.

Complexity of Reverse AD : Since we start with an output node for an arbitrary $f: \mathbb{R}^n \to \mathbb{R}^m$ and $\bar{u} \in \mathbb{R}^m$, Reverse AD computes $J^T\bar{u}$ in a single sweep in a matrix-free manner. The right choice of $\bar{u}$ would let us compute any row of J and consequently we would need m sweeps of Reverse AD to completely compute the Jacobian.

Since Reverse AD requires two graph traversals (Forward and Reverse) unlike Forward AD's single traversal, it has to deal with some performance overhead. For a single sweep of Reverse AD, [3]

$$OPS(f, J^T\bar{u}) \leq 4 \cdot OPS(f)$$

In ML applications e.g. neural nets $n >> m$ because n consists of thousands of weights and biases and m is just the output dimension of the neural network. And as, to compute the full Jacobian we need n sweeps of forward AD and m sweeps of backward / Reverse AD, in this case Reverse AD gives us superior performance.

Some frameworks built using AD

- In ML, Reverse AD is more popularly known as backpropagation which is the heart of a neural network.

- ADVI or Automatic Differentiation Variational Inference is a variational Bayesian inference framework that uses AD to maximize ELBO.

- Used in data assimilation/ inverse problems, Newton's method for solving non-linear equations, solving stiff ODEs.

References:

[1] On the equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation – Sören Laue (2020)

[2] A comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions -Rackauckas et al (2018)

[3] A review of Automatic Differentiation and its Efficient Implementation – Charles C. Margessian (2019)