Algorithmic Differentiation in More Depth

To delve a little more into AD, consider a computer implementation of a function f with n inputs and one output, i.e. $f : \mathbb{R}^n \to \mathbb{R}$. AD can be applied to vector-valued functions as well, but to keep things simple we only consider real valued functions below.

AD comes in two modes, forward and reverse.

Forward (tangent-linear) Mode AD

The forward (or tangent-linear) AD version of f is a function $F^{(1)} : \mathbb{R}^{2n} \to \mathbb{R}$ given by

$$y^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x^{(1)}}) = \nabla f(\mathbf{x}) \cdot \mathbf{x^{(1)}} = \left(\frac{\partial f}{\partial \mathbf{x}}\right) \cdot \mathbf{x^{(1)}}$$

for inputs $\mathbf{x}, \mathbf{x^{(1)}} \in \mathbb{R}^n$ where the dot is regular dot product. To get the whole gradient of f we let $\mathbf{x^{(1)}}$ range over Cartesian basis vectors and call $F^{(1)}$ repeatedly.

- The runtime of $F^{(1)}$ is typically similar to the runtime of f
- Computing the whole gradient is roughly n times the cost of computing f
- Forward mode AD has roughly the same cost as finite differences, but computes gradients to machine precision

Forward mode AD is typically used when n is small, say less than 30, alt
the exact figure will depend on the function being differentiated. Above t
adjoint methods are used.

Adjoint Mode AD (or reverse mode)

Intuition

To understand adjoint mode AD it helps to consider an input $\mathbf{x} \in \mathbb{R}^n$ being moved along by a sequence of function calls to an output $y \in \mathbb{R}$

$$\mathbf{x} \xrightarrow{f_1} \mathbf{x_1} \xrightarrow{f_2} \mathbf{x_2} \longrightarrow \cdots \longrightarrow \mathbf{x_m} \xrightarrow{f_{m+1}} y$$

We want the gradient $\partial y / \partial \mathbf{x}$ and by the Chain Rule this is just

$$\frac{\partial \mathbf{x_1}}{\partial \mathbf{x}} \frac{\partial \mathbf{x_2}}{\partial \mathbf{x_1}} \frac{\partial \mathbf{x_3}}{\partial \mathbf{x_2}} \cdots \frac{\partial \mathbf{x_m}}{\partial \mathbf{x_{m-1}}} \frac{\partial y}{\partial \mathbf{x_m}}$$

Mathematically

Mathematically it doesn't matter which way we evaluate this. The usual way is left to right

$$\left(\cdots \left(\frac{\partial \mathbf{x_1}}{\partial \mathbf{x}} \frac{\partial \mathbf{x_2}}{\partial \mathbf{x_1}} \right) \frac{\partial \mathbf{x_3}}{\partial \mathbf{x_2}} \right) \cdots \frac{\partial \mathbf{x_m}}{\partial \mathbf{x_{m-1}}} \bigg) \frac{\partial y}{\partial \mathbf{x_m}}$$

and this is natural since it corresponds to the order of program execution: the program first computes $\mathbf{x_1}$, then $\mathbf{x_2}$, and so on. However it involves **matrix-matrix multiplications** followed by final matrix-vector product since in general each Jacobian $\partial \mathbf{x_{i+1}} / \partial \mathbf{x_i}$ is a matrix.

Suppose instead we started from the right

$$\frac{\partial \mathbf{x_1}}{\partial \mathbf{x}} \left(\frac{\partial \mathbf{x_2}}{\partial \mathbf{x_1}} \left(\frac{\partial \mathbf{x_3}}{\partial \mathbf{x_2}} \cdots \left(\frac{\partial \mathbf{x_m}}{\partial \mathbf{x_{m-1}}} \frac{\partial y}{\partial \mathbf{x_m}} \right) \cdots \right) \right.\bigg)$$

Now everything is **matrix-vector products** which are much faster, however we *effectively need to run the program backwards*:

- The data to compute $\partial y / \partial \mathbf{x_m}$ is only available at the end of the calculation -- it requires y and $\mathbf{x_m}$, which requires $\mathbf{x_{m-1}}$, which requires $\mathbf{x_{m-2}}$ and so on
- One way to solve this is to run the program forwards and *store all relevant intermediate values*
- Then we step backwards, constructing the Jacobians $\frac{\partial \mathbf{x_{i+1}}}{\partial \mathbf{x_i}}$ from the stored values, and performing the matrix-vector products

This whole approach to computing gradients is called the *adjoint mode* of AD.

The Adjoint Model

The adjoint model of f is a function $\mathbf{x_{(1)}} = F_{(1)}(\mathbf{x}, \mathbf{x_{(1)}}, y_{(1)})$ mapping $\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}$ to $\mathbb{R}^n$ given by

$$\mathbf{x_{(1)}} = \mathbf{x_{(1)}} + \nabla f(\mathbf{x}) \cdot y_{(1)}$$

- Note that $y_{(1)}$ is a scalar. Hence setting $y_{(1)} = 1$ and $\mathbf{x_{(1)}} = 0$ and calling the adjoint model $F_{(1)}$ **once** gives the full vector of partial derivatives of f.
- The Jacobians $\partial \mathbf{x_{i+1}} / \partial \mathbf{x_i}$ are not formed explicitly and sparsity is exploited.
- It can be proved that, in general, computing $F_{(1)}$ requires no more than five times as many floating point operations as computing f.
- This implies that the adjoint can give the full gradient at a cost which is a (small) multiple $\mathcal{R}$ of the cost of running f.
- However to implement the adjoint model we need to solve a **dataflow reversal problem** which dominates by far the computational cost.
- Hence typical values of $\mathcal{R}$ are between 5 and 50, depending on the specific code.

Adjoint Model and Memory Requirements

Performing adjoint calculations requires solving a data flow reversal problem: the program essentially has to be run backwards. Many AD tools (including `dco`) approach this by running the program forwards and storing intermediate calculations to memory in a datastructure called a *tape*. Even for relatively simple codes the tape can be several GBs, and for production codes will typically exceed the capacity of even large memory machines.

To solve this problem, `dco` has a flexible interface which allows users to easily insert checkpoints at various points in their code. When the code is run backwards the final checkpoint is restored and that section of computation taped and played back, then the second-to-last checkpoint is restored and that section of computation is taped and played back (with the previous playback's results), and so on. In this way memory is traded for flops, with the result that the size of the tape can be constrained almost arbitrarily.

This functionality is essential in getting adjoint models of production cod
run at all. For more information on checkpointing as well as other techniq
of reducing the memory footprint of adjoint codes, please contact us
(/content/nag-technical-support-service#contact).

AD SOLUTIONS MORE INFORMATION →

Sign up for the NAG newsletter

SUBMIT →

ABOUT NAG (/CONTENT/ABOUT-NAG)

Blog (/content/nag-blog)
NAGnews (/content/nagnews-0)
Case Studies (/content/case-studies)
Contact us (/content/worldwide-contact-information)

SUPPORT (/CONTENT/TECHNICAL-SUPPORT-SERVICE-OVERVIEW)

Contact support (/content/technical-support-service-overview#contact)
Documentation (/content/software-documentation)
Installer's & Users' Notes (/content/installers-and-users-notes-nag-products)
Downloads (/content/software-downloads)
Technical Reports (/content/technical-report-repository)

twitter.com/nagtalk)	w.facebook.com/NAGTalk)	linkedin.com/company/nag/)	ube.com/user/NumericalAlgorithms)	om/numericalalgorithmsgroup)