Automatic Differentiation

Tobias Madsen

Introduction

Algorithms that use derivatives are ubiquitous in statistical inference. Automatic differentiation (AD) takes a function, $f : \mathbb{R}^M \to \mathbb{R}^N$, defined by a computer program and computes the derivatives analytically, doing so with the same computational complexity as the original program. Although AD deprives you of the satisfaction of deriving gradients by hand, it makes development faster and error rate lower.

Basic Idea

The basic idea of AD is to transform a computer program specifying a function into a directed acyclic graph called the computational graph. Each node of the graph corresponds to an intermediate variable and is the result of applying an *elemental function* to upstream nodes in the graph (Fig. 2). The elemental functions along with their partial derivatives are implemented in the system and should at a bare minimum include basic operations (Table 1). An AD tool for statistical inference should ideally also include a large set of density functions or their building blocks.

There are two modes of AD, forward and reverse. Forward-mode computes a directional derivative $J_f(c)\vec{x}$, reverse-mode computes $\vec{y}J_f(c)$. In forward-mode AD *input* takes a special role: For all downstream nodes in the computational graph we obtain a directional derivative w.r.t. the input nodes. In reverse-mode AD *output* takes a special role: For a specified output node we obtain the derivatives of that node w.r.t. all upstream nodes in the computational graph.

Reverse-mode is attractive in a situation where $f : \mathbb{R}^M \to \mathbb{R}$, e.g. a likelihood function with multiple parameters. Computing a gradient requires M passes with forward-mode AD, but only a single pass with reverse-mode AD (Fig. 1).

$$J_f(c) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Figure 1: A single pass of forward-mode AD gives a column in the Jacobian (or a linear combination of columns). A single pass of reverse-mode AD gives a row in the Jacobian, i.e. a gradient if $f : \mathbb{R}^M \to \mathbb{R}$.

Forward-mode AD

As the computational graph is a DAG we can order the nodes topologically, $v_1, \ldots, v_N$. Let the first K nodes be input nodes and let a direction vector $\vec{x} = (x_1, \ldots, x_K)$ be given. We iteratively

Domain	Functions
$\mathbb{R}^2$	$+, -, \cdot, /$
$\mathbb{R}$	$x \mapsto -x,\ x \mapsto c$
$\mathbb{R}$	$\exp, \log, \sin, \cos$

Table 1: Very minimal set of elemental functions.

compute the directional derivatives

$$t_i = \sum_{k=1}^{K} x_k \frac{\partial v_i}{\partial v_k}.$$

Note that to get a partial derivative with respect to a single variable simply use an indicator vector as direction vector, $\vec{x} = (0, \ldots, 0, 1, 0, \ldots, 0)$.

Forward AD proceeds by first initializing $t_i = x_i$ for $i = 1, \ldots K$. Next t_i is computed iteratively from $i = K + 1$ to $i = N$

$$\begin{aligned} t_i &= \sum_{k=1}^{K} x_k \frac{\partial v_i}{\partial v_k} \\ &= \sum_{k=1}^{K} x_k \sum_{j \in \text{Parents[i]}} \frac{\partial v_i}{\partial v_j} \frac{\partial v_j}{\partial v_k} \\ &= \sum_{j \in \text{Parents[i]}} \frac{\partial v_i}{\partial v_j} \sum_{k=1}^{K} x_k \frac{\partial v_j}{\partial v_k} \\ &= \sum_{j \in \text{Parents[i]}} \frac{\partial v_i}{\partial v_j} t_j. \end{aligned}$$

This can be done while evaluating the original expression, therefore there is no need to save intermediate results and explicitly build the computational graph.

There is a straightforward way of implementing forward AD using so-called *dual numbers*. A dual number consist of the value of the original expression at a node in the computational graph and its directional derivative. The basic operations and elemental functions must then respect both.

As an example consider the special case where the directional derivative just a regular partial derivative w.r.t. a variable h. To add to dual numbers $v = (v(h), v(h)')$ and $w = (w(h), w'(h))$, we simply have

$$\begin{aligned} v + w &= \left(v(h) + w(h),\ \frac{d}{dh} \left(v(h) + w(h) \right) \right) \\ &= \left(v(h) + w(h),\ v'(h) + w'(h) \right). \end{aligned}$$

Similarly to multiply to dual numbers

$$\begin{aligned} v * w &= \left(v(h)w(h),\ \frac{d}{dh} \left(v(h)w(h) \right) \right) \\ &= \left(v(h)w(h),\ v'(h)w(h) + v(h)w'(h) \right). \end{aligned}$$

That such a construction is generally possible is shown in [3] by first extending to polynomials and then any smooth function using Taylor polynomials. Fig. 6 shows a simple implementation of forward AD in `C++` using dual numbers and operator overloading.

Reverse-mode AD

In reverse mode AD we again consider a computational graph with the nodes topologically sorted. This time we pick a node v_N as a special output node. We then iteratively compute

$$a_i = \frac{\partial v_N}{\partial v_i}.$$

First, initialize $a_N = 1$. Second, iteratively from $i = N - 1$ to $i = 1$ compute

$$\begin{aligned} a_i &= \frac{\partial v_N}{\partial v_i} \\ &= \frac{\partial v_N(v_{N-1}, \ldots, v_i)}{\partial v_i} \\ &= \sum_{j=i}^{N-1} \frac{\partial v_N}{\partial v_j} \frac{\partial v_j}{\partial v_i} \\ &= \sum_{j \in \text{Children}[i]} a_j \frac{\partial v_j}{\partial v_i}. \end{aligned}$$

Symbolic and numeric differentiation

Numeric differentiation evaluates the gradient by finite differences. Each partial derivative is approximated by

$$\frac{\partial f}{\partial x_n}(x) \approx \frac{f(x_1, \ldots, x_n + \varepsilon, \ldots, x_M) - f(x_1, \ldots, x_M)}{\varepsilon}$$

for an $\varepsilon > 0$. Computing the full gradient requires $M + 1$ evaluations of f, contrasting AD where the computational complexity is essentially the same as a single evaluation of f. As ε approaches zero the above approximation approaches the true derivative, however for small ε we encounter problems with numerical stability caused by *catastrophic cancellation*.

Symbolic differentiation computes the entire derivative function analytically by applying the chain-rule. Let $f : \mathbb{R} \to \mathbb{R}$ be a composition of 3 elemental functions, $f = \phi_3 \circ \phi_2 \circ \phi_1$. Then symbolic differentiation first finds

$$f'(x) = \phi_3'(\phi_2(\phi_1(x)))\phi_2'(\phi_1(x))\phi_1'(x).$$

Then evaluates $f'(x)$ at c. This amounts to a total of 6 function evaluations. In general for a composition of k functions $\mathcal{O}(k^2)$ function evaluations is required[1]. Another advantage of AD over symbolic differentiation is that AD is more expressive than symbolic diffentiation, allowing for conditionals, for-loops and really any computational structure.

Implementations

Many libraries for AD exists[2]. Here I will focus on Stan Math and Tapenade. This is not the result of an extensive survey where these two tools were found to be superior, but mainly because their approach to AD is vastly different.

[1] I have only seen this claim in [3] and I suspect that there are implementations of symbolic differentiation that reuses computations.

[2] `http://www.autodiff.org/?module=Tools`

Stan Math

Stan math library [1] is a header only `C++` library used in the probabilistic programming framework Stan. Stan math contains a wide range of useful density functions. As Stan mostly needs gradient, Stan math has a working implementation of reverse-mode AD, but is still in the process of implementing forward mode. Fig. 3 shows an example of reverse-mode AD in Stan math, compilation depends on three additional libraries besides Stan math: Boost, Eigen, Cvodes, see `https://github.com/stan-dev/math` for more details.

Tapenade

Tapenade [2] uses static analysis to transform a program[3]. It takes as input a program written in `Fortran` or `C` and outputs a transformed program that computes not only the original function but also its derivatives. Takes a program written in C or Fortran and outputs a program with the derivative (Fig. 4 and 5. It implements both forward AD and reverse AD. A major advantage of program transformation is that it does not introduce any library dependencies for a potential end-user.

References

[1] Bob Carpenter, Matthew D Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. *arXiv preprint arXiv:1509.07164*, 2015.

[2] Laurent Hascoet and Valérie Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):20, 2013.

[3] Philipp HW Hoffmann. A Hitchhikers Guide to Automatic Differentiation. *Numerical Algorithms*, pages 1–37, 2015.

[3] Available online at `http://tapenade.inria.fr:8080/tapenade/`

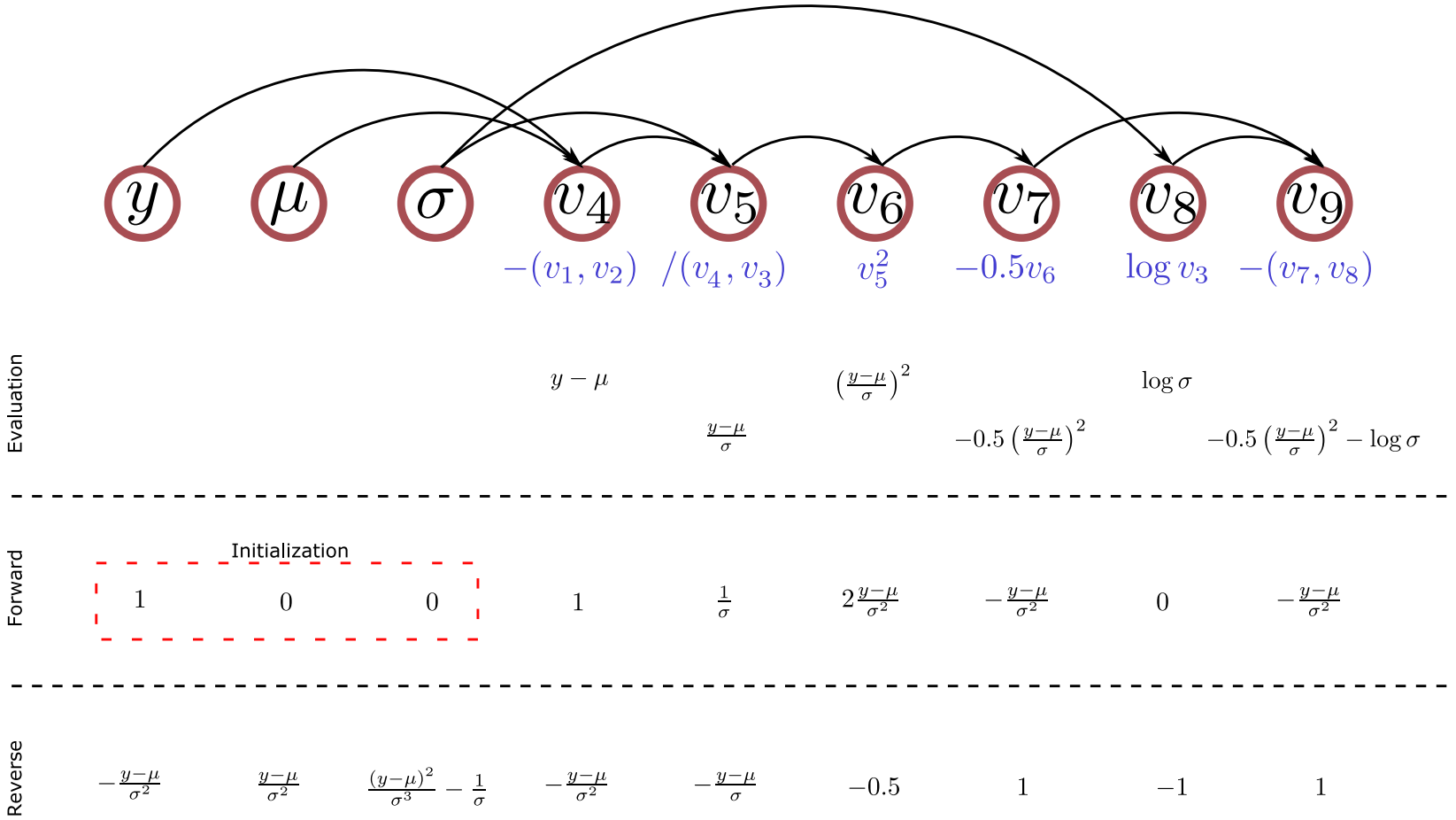

Figure 2: Computational graph for the function $-0.5 \cdot \left(\frac{y-\mu}{\sigma}\right)^2 - \log \sigma$

```
#include <stan/math.hpp>
#include <cmath>
#include <iostream>

int main() {
  stan::math::var y = 1.5;
  stan::math::var mu = 1.2;
  stan::math::var sigma = 0.5;
  stan::math::var lp = 0;

  lp += -0.5*pow( ((y-mu)/sigma) , 2) - log(sigma);

  std::cout << "lp:\t" << lp.val() << std::endl;

  lp.grad();

  std::cout << "dlp / dy:\t" << y.adj() << std::endl;
  std::cout << "dlp / dmu:\t" << mu.adj() << std::endl;
  std::cout << "dlp / dsigma:\t" << sigma.adj() << std::endl;
}
```

Figure 3: Example of using Stan math reverse-mode AD.

```
double normal(double y, double mu, double sigma, double ret){
  ret = -0.5 * pow((y-mu)/sigma, 2)-log(sigma);

  return ret;
}
```

Figure 4: Input to Tapenade

```
void normal_d(double y, double yd, double mu, double mud, double sigma, double
        sigmad, double ret, double retd, double *normal) {
    double arg1;
    double arg1d;
    arg1d = ((yd-mud)*sigma-(y-mu)*sigmad)/(sigma*sigma);
    arg1 = (y-mu)/sigma;
    retd = -(0.5*arg1d*2*pow(arg1, 2-1)) - sigmad/sigma;
    ret = -0.5*pow(arg1, 2) - log(sigma);
    *normal = ret;
}
```

Figure 5: Output from Tapenade in tangent (forward) mode.

```cpp
#include <iostream>
#include <cmath>

class Dual {
public:
  double x;
  double xd;

  // Constructors
  Dual() : x(0), xd(0) {} ;

  Dual(double x_, double xd_) : x(x_), xd(xd_) {} ;

  // For implicit conversion
  Dual(int x_) : x(x_), xd(0) {} ;
  Dual(double x_) : x(x_), xd(0) {} ;
};

// Operators
Dual operator-(Dual a){
  return Dual(-a.x, -a.xd);
}

Dual operator+(Dual a, Dual b){
  return Dual(a.x + b.x, a.xd + b.xd);
}

Dual operator-(Dual a, Dual b){
  return Dual(a.x - b.x, a.xd - b.xd);
}

Dual operator*(Dual a, Dual b){
  return Dual( a.x * b.x, a.xd * b.x + a.x * b.xd);
}

Dual operator/(Dual a, Dual b){
  return Dual(a.x / b.x, (a.xd * b.x - a.x * b.xd) / b.x / b.x);
}

// Overload ostream operator
std::ostream& operator<<(std::ostream& os, const Dual d){
  os << "(" << d.x << "," << d.xd << ")";
  return os;
}

// A few elemental function
Dual sin(Dual a){
  return Dual( std::sin(a.x), a.xd*std::cos(a.x));
}

Dual cos(Dual a){
  return Dual( std::cos(a.x), -a.xd*std::sin(a.x));
}

Dual exp(Dual a){
  return Dual( std::exp(a.x), a.xd*std::exp(a.x));
}

Dual log(Dual a){
  return Dual( std::log(a.x), a.xd / a.x);
}

Dual normalDensity(Dual y, Dual mu, Dual sigma){
  Dual a = (y - mu)/sigma;
  return -0.5*a*a-log(sigma);
}

int main(){
  // Derivative w.r.t y
  Dual y(1.5, 1);
  Dual mu(1.2, 0);
  Dual sigma(0.5, 0);
  std::cout << normalDensity(y, mu, sigma) << std::endl;

  // Derivative w.r.t sigma
  y = Dual(1.5,0);
  sigma = Dual(0.5,1);
  std::cout << normalDensity(y, mu, sigma) << std::endl;
}
```

Figure 6