8 Forward and Reverse-Mode Automatic Differentiation

The first time that Professor Edelman had heard about automatic differentiation (AD), it was easy for him to imagine what it was ... but what he imagined was wrong! In his head, he thought it was straightforward symbolic differentiation applied to code—sort of like executing Mathematica or Maple, or even just automatically doing what he learned to do in his calculus class. For instance, just plugging in functions and their domains from something like the following first-year calculus table:

Derivative	Domain
$(\sin x)' = \cos x$	$-\infty < x < \infty$
$(\cos x)' = -\sin x$	$-\infty < x < \infty$
$(\tan x)' = \sec^2 x$	$x \neq \frac{\pi}{2} + \pi n, n \in \mathbb{Z}$
$(\cot x)' = -\csc^2 x$	$x \neq \pi n, n \in \mathbb{Z}$
$(\sec x)' = \tan x \sec x$	$x \neq \frac{\pi}{2} + \pi n, n \in \mathbb{Z}$
$(\csc x)' = -\cot x \csc x$	$x \neq \pi n, n \in \mathbb{Z}$

And in any case, if it wasn't just like executing Mathematica or Maple, then it must be finite differences, like one learns in a numerical computing class (or as we did in Sec. 4).

It turns out that it is definitely *not* finite differences—AD algorithms are generally exact (in exact arithmetic, neglecting roundoff errors), not approximate. But it also doesn't look much like conventional symbolic algebra: the computer doesn't really construct a big "unrolled" symbolic expression and then differentiate it, the way you might imagine doing by hand or via computer-algebra software. For example, imagine a computer program that computes $\det A$ for an $n \times n$ matrix—writing down the "whole" symbolic expression isn't possible until the program runs and n is known (e.g. input by the user), and in any case a naive symbolic expression would require $n!$ terms. Thus, AD systems have to deal with computer-programming constructs like loops, recursion, and problem sizes n that are unknown until the program runs, while at the same time avoiding constructing symbolic expressions whose size becomes prohibitively large. (See Sec. 8.1.1 for an example that looks very different from the formulas you differentiate in first-year calculus.) Design of AD systems often ends up being more about compilers than about calculus!

8.1 Automatic Differentiation via Dual Numbers

(This lecture is accompanied by a Julia "notebook" showing the results of various computational experiments, which can be found on the course web page. Excerpts from those experiments are included below.)

One AD approach that can be explained relatively simply is "forward-mode" AD, which is implemented by carrying out the computation of f' in *tandem* with the computation of f. One augments every intermediate value a in the computer program with another value b that represents its derivative, along with chain rules to propagate these derivatives through computations on values in the program. It turns out that this can be thought of as replacing real numbers (values a) with a new kind of "dual number" $D(a, b)$ (values & derivatives) and corresponding arithmetic rules, as explained below.

8.1.1 Example: Babylonian square root

We start with a simple example, an algorithm for the square-root function, where a practical method of automatic differentiation came as both a mathematical surprise and a computing wonder for Professor Edelman. In particular, we consider the "Babylonian" algorithm to compute $\sqrt{x}$, known for millennia (and later revealed as a special case

of Newton's method applied to $t^2 - x = 0$): simply repeat $t \leftarrow (t + x/t)/2$ until t converges to $\sqrt{x}$ to any desired accuracy. Each iteration has one addition and two divisions. For illustration purposes, 10 iterations suffice. Here is a short program in Julia that implements this algorithm, starting with a guess of 1 and then performing N steps (defaulting to $N = 10$):

```
julia> function Babylonian(x; N = 10)
           t = (1+x)/2   # one step from t=1
           for i = 2:N   # remaining N-1 steps
               t = (t + x/t) / 2
           end
           return t
       end
```

If we run this function to compute the square root of $x = 4$, we will see that it converges very quickly: for only $N = 3$ steps, it obtains the correct answer (2) to nearly 3 decimal places, and well before $N = 10$ steps it has converged to 2 within the limits of the accuracy of computer arithmetic (about 16 digits). In fact, it roughly doubles the number of correct digits on every step:

```
julia> Babylonian(4, N=1)
2.5

julia> Babylonian(4, N=2)
2.05

julia> Babylonian(4, N=3)
2.000609756097561

julia> Babylonian(4, N=4)
2.0000000929222947

julia> Babylonian(4, N=10)
2.0
```

Of course, any first-year calculus student knows the derivative of the square root, $(\sqrt{x})' = 0.5/\sqrt{x}$, which we could compute here via `0.5 / Babylonian(x)`, but we want to know how we can obtain this derivative *automatically*, directly from the Babylonian algorithm itself. If we can figure out how to easily and efficiently pass the chain rule through this algorithm, then we will begin to understand how AD can also differentiate much more complicated computer programs for which no simple derivative formula is known.

8.1.2 Easy forward-mode AD

The basic idea of carrying the chain rule through a computer program is very simple: replace every number with *two* numbers, one which keeps track of the *value* and one which tracks the *derivative* of that value. The values are computed the same way as before, and the derivatives are computed by carrying out the chain rule for elementary operations like + and /.

In Julia, we can implement this idea by defining a new type of number, which we'll call `D`, that encapsulates a value `val` and a derivative `deriv`.

```
julia> struct D <: Number
           val::Float64
           deriv::Float64
       end
```

(A detailed explanation of Julia syntax can be found elsewhere, but hopefully you can follow the basic ideas even if you don't understand every punctuation mark.) A quantity `x = D(a,b)` of this new type has two components `x.val = a` and `x.deriv = b`, which we will use to represent values and derivatives, respectively. The `Babylonian` code only uses two arithmetic operations, + and /, so we just need to overload the built-in ("`Base`") definitions of these in Julia to include new rules for our `D` type:

```
julia> Base.:+(x::D, y::D) = D(x.val+y.val, x.deriv+y.deriv)
       Base.:/(x::D, y::D) = D(x.val/y.val, (y.val*x.deriv - x.val*y.deriv)/y.val^2)
```

If you look closely, you'll see that the values are just added and divided in the ordinary way, while the derivatives are computed using the sum rule (adding the derivatives of the inputs) and the quotient rule, respectively. We also need one other technical trick: we need to define "conversion" and "promotion" rules that tell Julia how to combine `D` values with ordinary real numbers, as in expressions like $x + 1$ or $x/2$:

```
julia> Base.convert(::Type{D}, r::Real) = D(r,0)
       Base.promote_rule(::Type{D}, ::Type{<:Real}) = D
```

This just says that an ordinary real number r is combined with a `D` value by first converting r to `D(r,0)`: the value is r and the derivative is 0 (the derivative of any constant is zero).

Given these definitions, we can now plug a `D` value into our *unmodified* `Babylonian` function, and it will "magically" compute the derivative of the square root. Let's try it for $x = 49 = 7^2$:

```
julia> x = 49
49

julia> Babylonian(D(x,1))
D(7.0, 0.07142857142857142)
```

We can see that it correctly returned a value of `7.0` and a derivative of `0.07142857142857142`, which indeed matches the square root $\sqrt{49}$ and its derivative $0.5/\sqrt{49}$:

```
julia> (√x, 0.5/√x)
(7.0, 0.07142857142857142)
```

Why did we input `D(x,1)`? Where did the 1 come from? That's simply the fact that the derivative of the input x with respect to *itself* is $(x)' = 1$, so this is the starting point for the chain rule.

In practice, all this (and more) has already been implemented in the ForwardDiff.jl package in Julia (and in many similar software packages in a variety of languages). That package hides the implementation details under the hood and explicitly provides a function to compute the derivative. For example:

```
julia> using ForwardDiff

julia> ForwardDiff.derivative(Babylonian, 49)
0.07142857142857142
```

Essentially, however, this is the same as our little `D` implementation, but implemented with greater generality and sophistication (e.g. chain rules for more operations, support for more numeric types, partial derivatives with respect to multiple variables, etc.): just as we did, ForwardDiff augments every value with a second number that tracks the derivative, and propagates both quantities through the calculation.

We could have also implemented the same idea specifically for the Bablylonian algorithm, by writing a new function `dBabylonian` that tracks both the variable t and its derivative $t' = dt/dx$ through the course of the calculation:

```
julia> function dBabylonian(x; N = 10)
           t = (1+x)/2
           t′ = 1/2
           for i = 1:N
               t = (t+x/t)/2
               t′= (t′+(t-x*t′)/t^2)/2
           end
           return t′
       end

julia> dBabylonian(49)
0.07142857142857142
```

This is doing *exactly* the same calculations as calling `Babylonian(D(x,1))` or `ForwardDiff.derivative(Babylonian, 49)`, but needs a lot more human effort—we'd have to do this for every computer program we write, rather than implementing a new number type *once*.

8.1.3 Dual numbers

There is a pleasing algebraic way to think about our new number type $D(a,b)$ instead of the "value & derivative" viewpoint above. Remember how a complex number $a + bi$ is formed from two real numbers (a,b) by defining a special new quantity i (the imaginary unit) that satisfies $i^2 = -1$, and all the other complex-arithmetic rules follow from this? Similarly, we can think of $D(a,b)$ as $a + b\epsilon$, where ϵ is a new "infinitesimal unit" quantity that satisfies $\epsilon^2 = 0$. This viewpoint is called a **dual number**.

Given the elementary rule $\epsilon^2 = 0$, the other algebraic rules for dual numbers immediately follow:

$$\begin{aligned}(a+b\epsilon) \pm (c+d\epsilon) &= (a \pm c) + (b \pm d)\epsilon \\ (a+b\epsilon) \cdot (c+d\epsilon) &= (ac) + (bc+ad)\epsilon \\ \frac{a+b\epsilon}{c+d\epsilon} &= \frac{a+b\epsilon}{c+d\epsilon} \cdot \frac{c-d\epsilon}{c-d\epsilon} = \frac{(a+b\epsilon)(c-d\epsilon)}{c^2} = \frac{a}{c} + \frac{bc-ad}{c^2}\epsilon.\end{aligned}$$

The ϵ coefficients of these rules correspond to the sum/difference, product, and quotient rules of differential calculus!

In fact, these are *exactly* the rules we implemented above for our `D` type. We were only missing the rules for subtraction and multiplication, which we can now include:

```
julia> Base.:-(x::D, y::D) = D(x.val - y.val, x.deriv - y.deriv)
       Base.:*(x::D, y::D) = D(x.val*y.val, x.deriv*y.val + x.val*y.deriv)
```

It's also nice to add a "pretty printing" rule to make Julia display dual numbers as `a + bϵ` rather than as `D(a,b)`:

```
julia> Base.show(io::IO, x::D) = print(io, x.val, " + ", x.deriv, "ϵ")
```

Once we implement the multiplication rule for dual numbers in Julia, then $\epsilon^2 = 0$ follows from the special case $a = c = 0$ and $b = d = 1$:

```
julia> ϵ = D(0,1)
0.0 + 1.0ϵ

julia> ϵ * ϵ
0.0 + 0.0ϵ

julia> ϵ^2
0.0 + 0.0ϵ
```

(We didn't define a rule for powers $D(a,b)^n$, so how did it compute ϵ^2? The answer is that Julia implements x^n via repeated multiplication by default, so it sufficed to define the $*$ rule.) Now, we can compute the derivative of the Babylonian algorithm at $x = 49$ as above by:

```
julia> Babylonian(x + ϵ)
7.0 + 0.07142857142857142ϵ
```

with the "infinitesimal part" being the derivative $0.5/\sqrt{49} = 0.0714\cdots$.

A nice thing about this dual-number viewpoint is that it corresponds directly to our notion of a derivative as linearization:

$$f(x+\epsilon) = f(x) + f'(x)\epsilon + (\text{higher-order terms}),$$

with the dual-number rule $\epsilon^2 = 0$ corresponding to dropping the higher-order terms.

8.2 Naive symbolic differentiation

Forward-mode AD implements the exact analytical derivative by propagating chain rules, but it is completely different from what many people *imagine* AD might be: evaluating a program *symbolically* to obtain a giant symbolic expression, and *then* differentiating this giant expression to obtain the derivative. A basic issue with this approach is that the size of these symbolic expressions can quickly explode as the program runs. Let's see what it would look like for the Babylonian algorithm.

Imagine inputting a "symbolic variable" x into our `Babylonian` code, running the algorithm, and writing a big algebraic expression for the result. After only one step, for example, we would get $(x+1)/2$. After two steps, we would get $((x+1)/2 + 2x/(x+1))/2$, which simplifies to a ratio of two polynomials (a "rational function"):

$$\frac{x^2 + 6x + 1}{4(x+1)}.$$

Continuing this process by hand is quite tedious, but fortunately the computer can do it for us (as shown in the accompanying Julia notebook). Three Babylonian iterations yields:

$$\frac{x^4 + 28x^3 + 70x^2 + 28x + 1}{8\left(x^3 + 7x^2 + 7x + 1\right)},$$

four iterations gives

$$\frac{x^8 + 120x^7 + 1820x^6 + 8008x^5 + 12870x^4 + 8008x^3 + 1820x^2 + 120x + 1}{16\left(x^7 + 35x^6 + 273x^5 + 715x^4 + 715x^3 + 273x^2 + 35x + 1\right)},$$

and five iterations produces the enormous expression:

$$\frac{x^{16}+496x^{15}+35960x^{14}+906192x^{13}+10518300x^{12}+64512240x^{11}+225792840x^{10}+471435600x^{9}+601080390x^{8}+471435600x^{7}+225792840x^{6}+64512240x^{5}+10518300x^{4}+906192x^{3}+35960x^{2}+496x+1}{32(x^{15}+155x^{14}+6293x^{13}+105183x^{12}+876525x^{11}+4032015x^{10}+10855425x^{9}+17678835x^{8}+17678835x^{7}+10855425x^{6}+4032015x^{5}+876525x^{4}+105183x^{3}+6293x^{2}+155x+1)}.$$

Notice how quickly these grow—in fact, the degree of the polynomials doubles on every iteration! Now, if we take the symbolic derivatives of these functions using our ordinary calculus rules, and simplify (with the help of the computer), the derivative of one iteration is $\frac{1}{2}$, of two iterations is

$$\frac{x^2+2x+5}{4\left(x^2+2x+1\right)},$$

of three iterations is

$$\frac{x^6+14x^5+147x^4+340x^3+375x^2+126x+21}{8\left(x^6+14x^5+63x^4+100x^3+63x^2+14x+1\right)},$$

of four iterations is

$$\frac{x^{14}+70x^{13}+3199x^{12}+52364x^{11}+438945x^{10}+2014506x^{9}+5430215x^{8}+8836200x^{7}+8842635x^{6}+5425210x^{5}+2017509x^{4}+437580x^{3}+52819x^{2}+3094x+85}{16(x^{14}+70x^{13}+1771x^{12}+20540x^{11}+126009x^{10}+440986x^{9}+920795x^{8}+1173960x^{7}+920795x^{6}+440986x^{5}+126009x^{4}+20540x^{3}+1771x^{2}+70x+1)},$$

and of five iterations is a monstrosity you can only read by zooming in:

This is a terrible way to compute derivatives! (However, more sophisticated approaches to efficient symbolic differentiation exist, such as the "D^*" algorithm, that avoid explicit giant formulas by exploiting repeated subexpressions.)

To be clear, the dual number approach (absent rounding errors) computes an answer exactly as if it evaluated these crazy expressions at some particular x, but the words "as if" are very important here. As you can see, we do not form these expressions, let alone evaluate them. We merely compute results that are equal to the values we would have gotten if we had.

8.3 Automatic Differentiation via Computational Graphs

Let's now get into automatic differentiation via computational graphs. For this section, we consider the following simple motivating example.

Example 40

Define the following functions:

$$\begin{cases} a(x,y) = \sin x \\ b(x,y) = \frac{1}{y} \cdot a(x,y) \\ z(x,y) = b(x,y) + x. \end{cases}$$

Compute $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$.

There are a few ways to solve this problem. Firstly, of course, one can compute this symbolically, noting that

$$z(x,y) = b(x,y) + x = \frac{1}{y}a(x,y) + x = \frac{\sin x}{y} + x,$$

which implies

$$\frac{\partial z}{\partial x} = \frac{\cos x}{y} + 1 \quad \text{and} \quad \frac{\partial z}{\partial y} = -\frac{\sin x}{y^2}.$$

However, one can also use a Computational Graph (see Figure of Computational Graph below) where the edge from node A to node B is labelled with $\frac{\partial B}{\partial A}$.

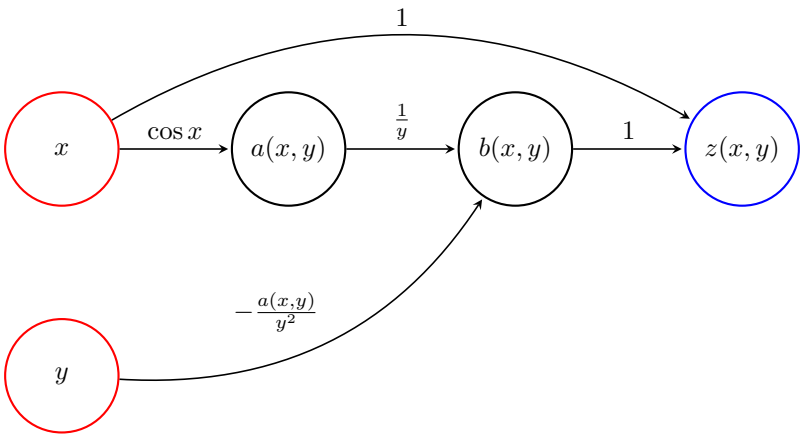


Figure 7: A computational graph corresponding to example 40, representing the computation of an output $z(x,y)$ from two inputs x, y, with intermediate quantities $a(x,y)$ and $b(x,y)$. The nodes are labelled by *values*, and edges are labelled with the *derivatives* of the values with respect to the preceding values.

Now how do we use this directed acyclic graph (DAG) to find the derivatives? Well one view (called the "forward view") is given by following the paths from the inputs to the outputs and (left) multiplying as you go, adding together multiple paths. For instance, following this procedure for paths from x to $z(x,y)$, we have

$$\frac{\partial z}{\partial x} = 1 \cdot \frac{1}{y} \cdot \cos x + 1 = \frac{\cos x}{y} + 1.$$

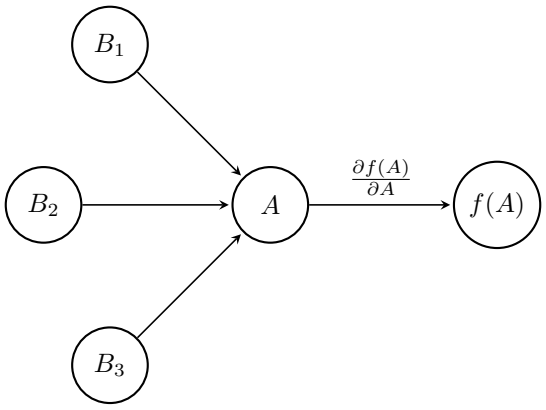
Similarly, for paths from y to $z(x, y)$, we have

$$\frac{\partial z}{\partial y} = 1 \cdot \frac{-a(x,y)}{y^2} = \frac{-\sin x}{y^2},$$

and if you have numerical derivatives on the edges, this algorithm works. Alternatively, you could follow a reverse view and follow the paths backwards (multiplying right to left), and obtain the same result. Note that there is nothing magic about these being scalar here– you could imagine these functions are the type that we are seeing in this class and do the same computations! The only thing that matters here fundamentally is the associativity. However, when considering vector-valued functions, the order in which you multiply the edge weights is vitally important (as vector/matrix valued functions are not generally commutative).

The graph-theoretic way of thinking about this is to consider "path products". A path product is the product of edge weights as you traverse a path. In this way, we are interested in the sum of path products from inputs to outputs to compute derivatives using computational graphs. Clearly, we don't particularly care which order we traverse the paths as long as the *order* we take the product in is correct. In this way, forward and reverse-mode automatic differentiation is not so mysterious.

Let's take a closer view of the implementation of forward-mode automatic differentiation. Suppose we are at a node A during the process of computing the derivative of a computational graph, as shown in the figure below:



Suppose we know the path product P of all the edges up to and including the one from B_2? to A. Then what is the new path product as we move to the right from A? It is $f'(A) \cdot P$! So we need a data structure that maps in the following way:

$$(\text{value}, \text{path product}) \mapsto (f(\text{value}), f' \cdot \text{path product}).$$

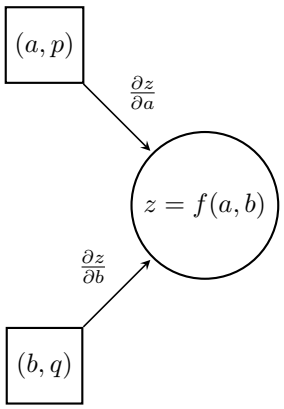
In some sense, this is another way to look at the Dual Numbers– taking in our path products and spitting out values. In any case, we overload our program which can easily calculate $f(\text{value})$ and tack-on $f' \cdot (\text{path product})$.

One might ask how our program starts– this is how the program works in the "middle", but what should our starting value be? Well the only thing it can be for this method to work is $(x, 1)$. Then, at every step you do the following map listed above:

$$(\text{value}, \text{path product}) \mapsto (f(\text{value}), f' \cdot \text{path product}),$$

and at the end we obtain our derivatives.

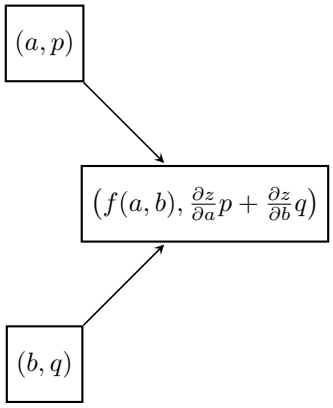
Now how do we combine arrows? In other words, suppose at the two notes on the LHS we have the values (a, p) and (b, q), as seen in the diagram below: So here, we aren't thinking of a, b as numbers, but as variables. What



should the new output value be? We want to add the two path products together, obtaining

$$\left(f(a,b), \frac{\partial z}{\partial a}p + \frac{\partial z}{\partial b}q\right).$$

So really, our overloaded data structure looks like this:



This diagram of course generalizes if we may many different nodes on the left side of the graph.

If we come up with such a data structure for all of the simple computations (addition/subtraction, multiplication, and division), and if this is all we need for our computer program, then we are set! Here is how we define the structure for addition/subtraction, multiplication, and division.

Addition/Subtraction: See figure.

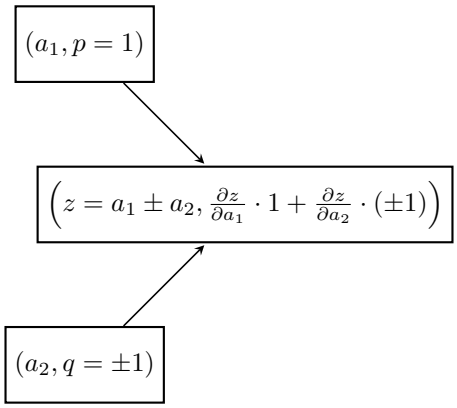


Figure 8: Figure of Addition/Subtraction Computational Graph

Multiplication: See figure.

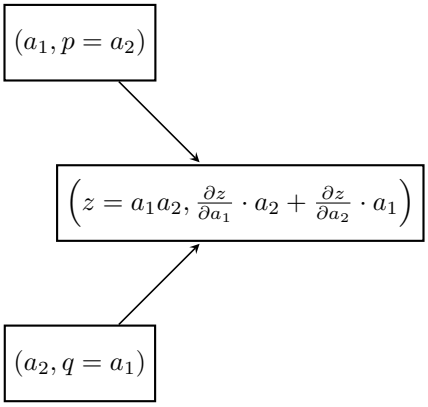


Figure 9: Figure of Multiplication Computational Graph

Division: See figure.

In theory, these three graphs are all we need, and we can use Taylor series expansions for more complicated functions. But in practice, we throw in what the derivatives of more complicated functions are so that we don't waste our time trying to compute something we already know, like the derivative of sine or of a logarithm.

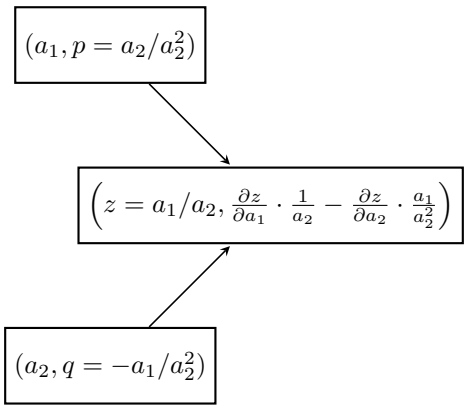
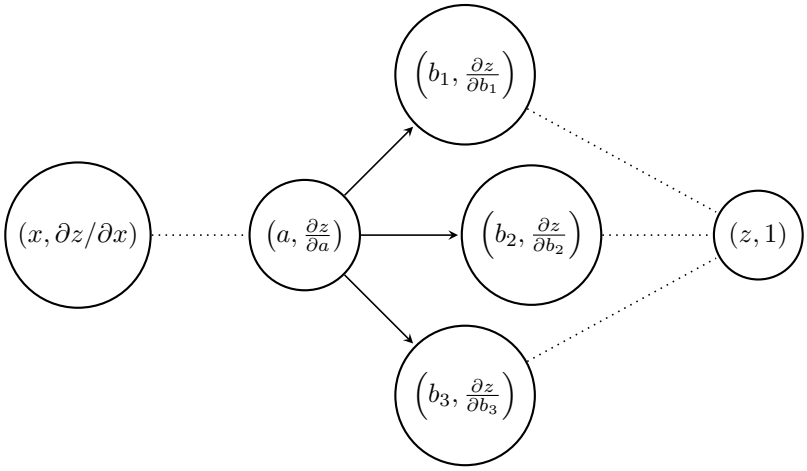


Figure 10: Figure of Division Computational Graph

8.3.1 Reverse Mode Automatic Differentiation on Graphs

When we do reverse mode, we have arrows going the other direction, which we will understand in this section of the notes. In forward mode it was all about "what do we depend on," i.e. computing the derivative on the right hand side of the above diagram using the functions in the nodes on the left. In reverse mode, the question is really "what are we influenced by?" or "what do we influence later?"

When going "backwards," we need know what nodes a given node influences. For instance, given a node A, we want to know the nodes B_i that is influenced by, or depends on, node A. So now our diagram looks like this:

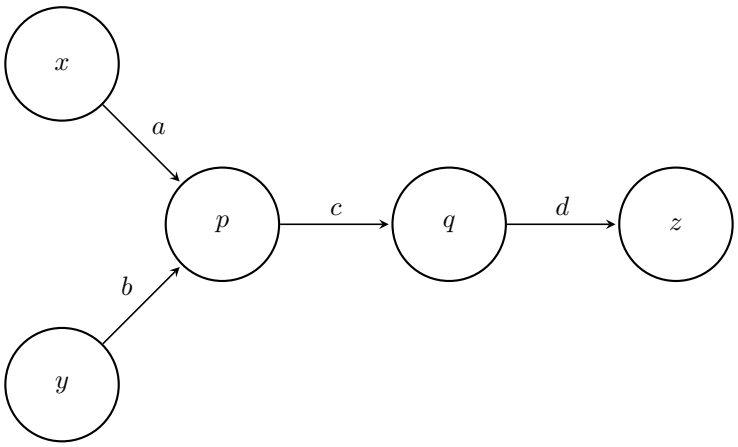


So now, we eventually have a final node $(z, 1)$ (far on the right hand side) where everything starts. This time, all of our multiplications take place from right to left as we are in reverse mode. Our goal is to be able to calculate the node $(x, \partial z/\partial x)$. So if we know how to fill in the $\frac{\partial z}{\partial a}$ term, we will be able to go from right to left in these computational graphs (i.e., in reverse mode). In fact, the formula for getting $\frac{\partial z}{\partial a}$ is given by

$$\frac{\partial z}{\partial a} = \sum_{i=1}^{s} \frac{\partial b_i}{\partial a} \frac{\partial z}{\partial b_i}$$

where the b_is come from the nodes that are influenced by the node A. This is again just another chain rule like from calculus, but you can also view this as multiplying the sums of all the weights in the graph influenced by A.

Why can reverse mode be more efficient than forward mode? One reason it because it can save data and use it



later. Take, for instance, the following sink/source computational graph.

If x, y here are our sources, and z is our sink, we want to compute the sum of products of weights on paths from sources to sinks. If we were using forward mode, we would need to compute the paths dca and dcb, which requires four multiplications (and then you would add them together). If we were using reverse mode, we would only need compute $a\underline{cd}$ and $b\underline{cd}$ and sum them; notice reverse mode (since we need only compute cd once), only takes 3 multiplications. In general, this can more efficiently resolve certain types of problems, such as the source/sink one.

8.4 Forward- vs. Reverse-mode Differentiation

In this section, we briefly summarize the relative benefits and drawbacks of these two approaches to computation of derivatives (whether worked out by hand or using AD software). From a mathematical point of view, the two approaches are mirror images, but from a computational point of view they are quite different, because computer programs normally proceed "forwards" in time from inputs to outputs.

Suppose we are differentiating a function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$, mapping n scalar inputs (an n-dimensional input) to m scalar outputs (an m-dimensional output). The first key distinction of forward- vs. reverse-mode is how the computational cost scales with the number/dimension of inputs and outputs:

- The cost of forward-mode differentiation (inputs-to-outputs) scales proportional to n, the number of *inputs*. This is ideal for functions where $n \ll m$ (few inputs, many outputs).
- The cost of reverse-mode differentiation (outputs-to-inputs) scales proportional to m, the number of *outputs*. This is ideal for functions where $m \ll n$ (few outputs, many inputs).

Before this chapter, we first saw these scalings in Sec. 2.5.1, and again in Sec. 6.3; in a future lecture, we'll see it yet again in Sec. 9.2. The case of few outputs is extremely common in large-scale optimization (whether for machine learning, engineering design, or other applications), because then one has many optimization parameters ($n \gg 1$) but only a single output ($m = 1$) corresponding to the objective (or "loss") function, or sometimes a few outputs corresponding to objective and constraint functions. Hence, reverse-mode differentiation ("backpropagation") is the dominant approach for large-scale optimization and applications such as training neural networks.

There are other practical issues worth considering, however:

- Forward-mode differentiation proceeds in the same order as the computation of the function itself, from inputs to outputs. This seems to make forward-mode AD easier to implement (e.g. our sample implementation in Sec. 8.1) and efficient.

- Reverse-mode differentiation proceeds in the *opposite* direction to ordinary computation. This makes reverse-mode AD much more complicated to implement, and adds a lot of *storage overhead* to the function computation. First you evaluate the function from inputs to outputs, but you (or the AD system) keep a *record* (a "tape") of all the *intermediate steps* of the computation; then, you run the computation in *reverse* ("play the tape backwards") to backpropagate the derivatives.

As a result of these practical advantages, even for the case of many ($n > 1$) inputs and a single ($m = 1$) output, practitioners tell us that they've found forward mode to be more efficient until n becomes sufficiently large (perhaps even until $n > 100$, depending on the function being differentiated and the AD implementation). (You may also be interested in the blog post Engineering Trade-offs in AD by Chris Rackauckas, which is mainly about reverse-mode implementations.)

If $n = m$, where neither approach has a scaling advantage, one typically prefers the lower overhead and simplicity of forward-mode differentiation. This case arises in computing explicit Jacobian matrices for nonlinear root-finding (Sec. 6.1), or Hessian matrices of second derivatives (Sec. 12), for which one often uses forward mode... or even a *combination* of forward and reverse modes, as discussed below.

Of course, forward and reverse are not the only options. The chain rule is associative, so there are many possible orderings (e.g. starting from both ends and meeting in the middle, or vice versa). A difficult[6] problem that may often require hybrid schemes is to compute Jacobians (or Hessians) in a minimal number of operations, exploiting any problem-specific structure (e.g. sparsity: many entries may be zero). Discussion of this and other AD topics can be found, in vastly greater detail than in these notes, in the book *Evaluating Derivatives* (2nd ed.) by Griewank and Walther (2008).

8.4.1 Forward-over-reverse mode: Second derivatives

Often, a *combination* of forward- and reverse-mode differentiation is advantageous when computing *second* derivatives, which arise in many practical applications.

Hessian computation: For example, let us consider a function $f(x) : \mathbb{R}^n \to \mathbb{R}$ mapping n inputs x to a single scalar. The first derivative $f'(x) = (\nabla f)^T$ is best computed by reverse mode if $n \gg 1$ (many inputs). Now, however, consider the *second* derivative, which is the derivative of $g(x) = \nabla f$, mapping n inputs x to n outputs ∇f. It should be clear that $g'(x)$ is therefore an $n \times n$ Jacobian matrix, called the **Hessian** of f, which we will discuss much more generally in Sec. 12. Since $g(x)$ has the same number of inputs and outputs, neither forward nor reverse mode has an inherent scaling advantage, so typically forward mode is chosen for g' thanks to its practical simplicity, while still computing ∇f in reverse-mode. That is, we compute ∇f by reverse mode, but then compute $g' = (\nabla f)'$ by applying forward-mode differentiation to the ∇f algorithm. This is called a **forward-over-reverse** algorithm.

An even more clear-cut application of forward-over-reverse differentiation is to **Hessian–vector products**. In many applications, it turns out that what is required is only the *product* $(\nabla f)'v$ of the Hessian $(\nabla f)'$ with an arbitrary vector v. In this case, one can completely avoid computing (or storing) the Hessian matrix explicitly, and incur computational cost proportional only to that of a single function evaluation $f(x)$. The trick is to recall (from Sec. 2.2.1) that, for *any* function g, the linear operation $g'(x)[v]$ is a *directional derivative*, equivalent to a *single-variable* derivative $\frac{\partial}{\partial \alpha} g(x + \alpha v)$ evaluated at $\alpha = 0$. Here, we simply apply that rule to the function $g(x) = \nabla f$, and obtain the following formula for a Hessian–vector product:

$$(\nabla f)'v = \frac{\partial}{\partial \alpha} \left(\nabla f|_{x+\alpha v} \right) \Big|_{\alpha=0} .$$

[6] In fact, extraordinarily difficult: "NP-complete" (Naumann, 2006).

Computationally, the inner evaluation of the gradient ∇f at an arbitrary point $x+\alpha v$ can be accomplished efficiently by a reverse/adjoint/backpropagation algorithm. In contrast, the *outer* derivative with respect to a *single* input α is best performed by forward-mode differentiation.[7] Since the Hessian matrix is symmetric (as discussed in great generality by Sec. 12), the same algorithm works for **vector–Hessian products** $v^T(\nabla f)' = [(\nabla f)'v]^T$, a fact that we employ in the next example.

Scalar-valued functions of gradients: There is another common circumstance in which one often combines forward and reverse differentiation, but which can appear somewhat more subtle, and that is in differentiating a scalar-valued function of a gradient of another scalar-valued function. Consider the following example:

Example 41

Let $f(x) : \mathbb{R}^n \mapsto \mathbb{R}$ be a scalar-valued function of $n \gg 1$ inputs with gradient $\nabla f|_x = f'(x)^T$, and let $g(z) : \mathbb{R}^n \mapsto \mathbb{R}$ be another such function with gradient $\nabla g|_z = g'(z)^T$. Now, consider the scalar-valued function $h(x) = g(\nabla f|_x) : \mathbb{R}^n \mapsto \mathbb{R}$ and compute $\nabla h|_x = h'(x)^T$.

Denote $z = \nabla f|_x$. By the chain rule, $h'(x) = g'(z)(\nabla f)'(x)$, but we want to avoid explicitly computing the large $n \times n$ Hessian matrix $(\nabla f)'$. Instead, as discussed above, we use the fact that such a vector–Hessian product is equivalent (by symmetry of the Hessian) to the transpose of a Hessian–vector product multiplying the Hessian $(\nabla f)'$ with the vector $\nabla g = g'(z)^T$, which is equivalent to a directional derivative:

$$\nabla h|_x = h'(x)^T = \frac{\partial}{\partial \alpha} \left(\nabla f|_{x+\alpha \nabla g|_z} \right) \Big|_{\alpha=0},$$

involving differentiation with respect to a single scalar $\alpha \in \mathbb{R}$. As for any Hessian–vector product, therefore, we can evaluate h and ∇h by:

1. Evaluate $h(x)$: evaluate $z = \nabla f|_x$ by reverse mode, and plug it into $g(z)$.
2. Evaluate ∇h:
   (a) Evaluate $\nabla g|_z$ by reverse mode.
   (b) Implement $\nabla f|_{x+\alpha \nabla g|_z}$ by reverse mode, and then differentiate with respect to α by *forward* mode, evaluated at $\alpha = 0$.

This is a "forward-over-reverse" algorithm, where forward mode is used efficiently for the single-input derivative with respect to $\alpha \in \mathbb{R}$, combined with reverse mode to differentate with respect to $x, z \in \mathbb{R}^n$.

Example Julia code implementing the above "forward-over-reverse" process for just such a $h(x) = g(\nabla f)$ function is given below. Here, the forward-mode differentiation with respect to α is implemented by the ForwardDiff.jl package discussed in Sec. 8.1, while the reverse-mode differentiation with respect to x or z is performed by the Zygote.jl package. First, let's import the packages and define simple example functions $f(x) = 1/\|x\|$ and $g(z) = (\sum_k z_k)^3$, along with the computation of h via Zygote:

```
julia> using ForwardDiff, Zygote, LinearAlgebra
julia> f(x) = 1/norm(x)
julia> g(z) = sum(z)^3
julia> h(x) = g(Zygote.gradient(f, x)[1])
```

[7] The Autodiff Cookbook, part of the JAX documentation, discusses this algorithm in a section on Hessian–vector products. It notes that one could also interchange the $\partial/\partial\alpha$ and ∇_x derivatives and employ reverse-over-forward mode, but suggests that this is less efficient in practice: "because forward-mode has less overhead than reverse-mode, and since the outer differentiation operator here has to differentiate a larger computation than the inner one, keeping forward-mode on the outside works best." It also presents another alternative: using the identity $(\nabla f)'v = \nabla(v^T \nabla f)$, one can apply reverse-over-reverse mode to take the gradient of $v^T \nabla f$, but this has even more computational overhead.

Now, we'll compute ∇h by forward-over-reverse:

```
julia> function ∇h(x)
           ∇f(y) = Zygote.gradient(f, y)[1]
           ∇g = Zygote.gradient(g, ∇f(x))[1]
           return ForwardDiff.derivative(α -> ∇f(x + α*∇g), 0)
       end
```

We can now plug in some random numbers and compare to a finite-difference check:

```
julia> x = randn(5); δx = randn(5) * 1e-8;

julia> h(x)
-0.005284687528953334

julia> ∇h(x)
5-element Vector{Float64}:
 -0.006779692698531759
  0.007176439898271982
 -0.006610264199241697
 -0.0012162087082746558
  0.007663756720005014

julia> ∇h(x)' * δx    # directional derivative
-3.0273434457397667e-10

julia> h(x+δx) - h(x)  # finite-difference check
-3.0273433933303284e-10
```

The finite-difference check matches to about 7 significant digits, which is as much as we can hope for—the forward-over-reverse code works!

Problem 42

A common variation on the above procedure, which often appears in machine learning, involves a function $f(x, p) \in \mathbb{R}$ that maps input "data" $x \in \mathbb{R}^n$ and "parameters" $p \in \mathbb{R}^N$ to a scalar. Let $\nabla_x f$ and $\nabla_p f$ denote the gradients with respect to x and p.

Now, suppose we have a function $g(z) : \mathbb{R}^n \mapsto \mathbb{R}$ as before, and define $h(x, p) = g(\nabla_x f|_{x,p})$. We want to compute $\nabla_p h = (\partial h/\partial p)^T$, which will involve "mixed" derivatives of f with respect to *both* x and p.

Show that you can compute $\nabla_p h$ by:

$$\nabla_p h|_{x,p} = \frac{\partial}{\partial \alpha} \left(\nabla_p f|_{x+\alpha \nabla g|_z, p} \right) \Big|_{\alpha=0},$$

where $z = \nabla_x f|_{x,p}$. (Crucially, this avoids ever computing an $n \times N$ mixed-derivative matrix of f.)

Try coming up with simple example functions f and g, implementing the above formula by forward-over-reverse in Julia similar to above (forward mode for $\partial/\partial\alpha$ and reverse mode for the ∇'s), and checking your result against a finite-difference approximation.

18.S096 Matrix Calculus for Machine Learning and Beyond
Independent Activities Period (IAP) 2023