Automatic Differentiation

David Barber

What is AutoDiff?

- AutoDiff takes a function $f(\mathbf{x})$ and returns an exact value (up to machine accuracy) for the gradient

$$g_i(\mathbf{x}) \equiv \frac{\partial}{\partial x_i} f \Big|_{\mathbf{x}}$$

- Note that this is not the same as a numerical approximation (such as central differences) for the gradient.
- One can show that, if done efficiently, one can always calculate the gradient in less than 5 times the time it takes to compute $f(\mathbf{x})$.
- This is also *not* the same as symbolic differentiation.

Symbolic Differentiation

- Given a function $f(x) = \sin(x)$, symbolic differentiation returns an algebraic expression for the derivative. This is not necessarily efficient since it may contain a great number of terms.
- As an (overly!) simple example, consider

  $$f(x_1, x_2) = \left(x_1^2 + x_2^2\right)^2$$

  $$\frac{\partial f}{\partial x_1} = 2\left(x_1^2 + x_2^2\right) 2x_1, \qquad \frac{\partial f}{\partial x_2} = 2\left(x_1^2 + x_2^2\right) 2x_2$$

  The algebraic expression is not computationally efficient. However, by defining $y = 4(x_1^2 + x_2^2)$,

  $$\frac{\partial f}{\partial x_1} = yx_1, \qquad \frac{\partial f}{\partial x_2} = yx_2$$

  Which is a more efficient *computational* expression.
- Also, more generally, we want to consider computational subroutines that contain loops and conditional `if` statements; these do not correspond to simple closed algebraic expressions. We want to find a corresponding subroutine that can return the exact derivative efficiently for such subroutines.

Forward and Reverse Differentiation

Forward

- This is (usually) easy to implement
- However, it is not (generally) computationally efficient.
- It cannot easily handle conditional statements or loops.

Reverse

- This is exact and computationally efficient.
- It is, however, harder to code and requires a parse tree of the subroutine.
- If possible, one should always attempt to do reverse differentiation.
- As we will discuss, the famous backprop algorithm is just a special case of reverse differentiation.
- Reverse differentiation is also important since, with it, one can understand (for example) how to deal easily with calculating the derivative of a function subject to parameter tying.

Forward Differentiation

Consider $f(x) = x^2$.

Complex arithmetic

$$f(x + i\epsilon) = (x + i\epsilon)^2 = x^2 - \epsilon^2 + 2i\epsilon x$$

$$f'(x) = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \mathrm{Im}\left(f(x + i\epsilon)\right)$$

- This also holds for any smooth function (one that an be expressed as a Taylor series).
- For finite ϵ this gives an *approximation* only.
- More accurate approximation than standard finite differences since we do not subtract two small quantities and divide by a small quantity – the complex arithmetic approach is more numerically stable.
- To implement, we need to overload all functions so that they can deal with complex arithmetic.

Forward Differentiation

Consider $f(x) = x^2$.

Dual arithmetic

Define an idempotent variable, ϵ such that $\epsilon^2 = 0$.

$$f(x+\epsilon) = (x+\epsilon)^2 = x^2 + 2x\epsilon$$

Hence

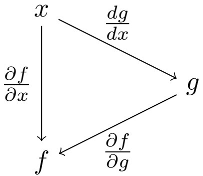
$$f'(x) = \mathsf{DualPart} f(x+\epsilon)$$

- This holds for any smooth function $f(x)$ and non-zero value of ϵ.
- Need to overload every function in the subroutine to work in dual arithmetic.
- Numerically *exact*.
- Whilst exact, this is not necessarily efficient.

Reverse Differentiation

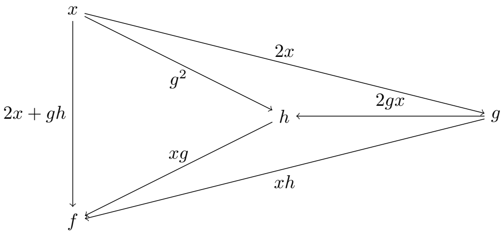
A useful graphical representation is that the total derivative of f with respect to x is given by the sum over all path values from x to f, where each path value is the product of the partial derivatives of the functions on the edges:

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial g}\frac{dg}{dx}$$



Example

For $f(x) = x^2 + xgh$, where $g = x^2$ and $h = xg^2$



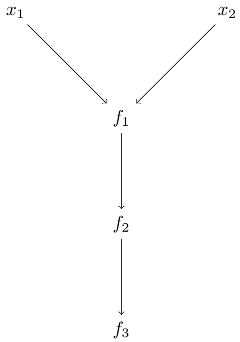
$$f'(x) = (2x + gh) + (g^2xg) + (2x2gxxg) + (2xxh) = 2x + 8x^7$$

Reverse Differentiation

Consider

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

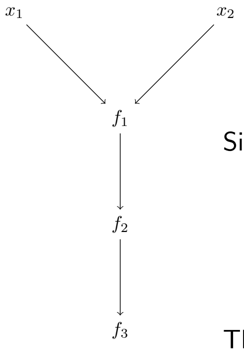
We can represent this computationally using an Abstract Syntax Tree (AST):



$$\begin{aligned} f_1(x_1, x_2) &= x_1 x_2 \\ f_2(x) &= \sin(x) \\ f_3(x) &= \cos(x) \end{aligned}$$

Given values for x_1, x_2, we first run forwards through the tree so that we can associate each node with an actual function value.

Reverse Differentiation



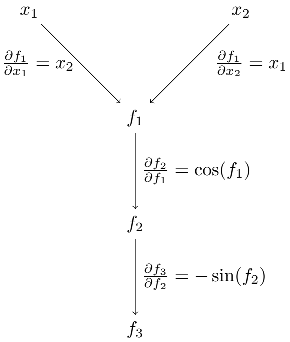
$$\frac{df_3}{dx_1} = \frac{\partial f_3}{\partial f_2}\frac{df_2}{dx_1} = \underbrace{\frac{\partial f_3}{\partial f_2}\frac{df_2}{df_1}}_{\frac{df_3}{df_1}}\frac{df_1}{dx_1}$$

Similarly,

$$\frac{df_3}{dx_2} = \underbrace{\frac{\partial f_3}{\partial f_2}\frac{df_2}{df_1}}_{\frac{df_3}{df_1}}\frac{df_1}{dx_2}$$

The two derivatives share the same computation branch and we want to exploit this.

Reverse Differentiation



1. Find the reverse ancestral (backwards) schedule of nodes $(f_3, f_2, f_1, x_1, x_2)$.
2. Start with the first node n_1 in the reverse schedule and define $t_{n_1} = 1$.
3. For the next node n in the reverse schedule, find the child nodes $\mathrm{ch}\,(n)$. Then define

$$t_n = \sum_{c \in \mathrm{ch}(n)} \frac{\partial f_c}{\partial f_n} t_c$$

4. The total derivatives of f with respect to the root nodes of the tree (here x_1 and x_2) are given by the values of t at those nodes.

This is a general procedure that can be used to automatically define a subroutine to efficiently compute the gradient. It is efficient because information is collected at nodes in the tree and split between parents only when required.

Dealing with loops

```
f=function(x)
f=0;
for i=1:10
.   f=f+cos(f*x^i);
end
```

```
df=function(x)
f=0;
df=0;
for i=1:10
.   f=f+cos(f*x^i);
.   df=df-sin(f*x^i)*(f*i*x^(i-1)+df*x^i);
end
```

- Above we expanded the derivative of the $\cos$ term symbolically.
- In AutoDiff we would replace this step with the computations on the AST.

Software

- AutoDiff has been around a long time (since the 1960's).
- There are tons of tools out there with varying degrees of sophistication.
- The most efficient tools use special purpose optimisers to first obtain the most compact AST.
- Stan is a popular recent C++ tools from Stanford.
- Theano is a popular tool in python, developed by Montreal Machine Learners.