Batch Normalization Notes

Piyush Patil

September 19, 2017

1 Introduction

Automatic differentiation is a technique to numerically compute the derivative of a function implemented by a computer program. It is distinct from numerical differentiation, which uses often iterative approximation techniques to converge to a guess of the numerical value of a derivative within some error margine, and from symbolic differentiation, which executes elementary differentiation by implementing the mechanical algorithms arising from common differentiation rules and techniques of calculus, such as the power rule or chain rule. Both methods suffer from shortcomings. Numerical differentiation is prone to roundoff errors which further compound and propagate when computing higher-order derivatives or derivatives of complex, highly composed functions. Symbolic differentiation often leads to very complex, convoluted expression trees that are inefficient to evaluate, unless properly repaired by a simplification module, which itself requires non-trivial computation and is difficult to implement well. Moreover, both operations tend to have linear time complexity in terms of the dimensionality of the function being differentiated.

Automatic differentiation, on the other hand, typically requires only a constant factor more operations to compute relative to the computation of the program (which implements the function) itself. It exploits the fact that any function implemented on a computer (or, more abstractly, a Turing machine) ultimately is executing a sequence of elementary operations or computing elementary functions. Automatic differentiation uses the chain rule to hierarchically compute derivatives using these elementary operations, and is able to do so to arbitrary precision.

2 Forward Mode vs Backward Mode

Automatic differentiation operates on the directed acyclic graph of expressions representing a function that's decomposed into compositions of elementary operations and functions, i.e. into a graph whose nodes are elementary functions (including operations such as addition and multiplication) and whose edges are variables that are fed as input into functions. There are two ways of implementing automatic differentiation, referred to as forward mode and backward mode, which respectively correspond to bottom-up and top-down applications of the chain rule to the expression DAG.

Given an expression tree whose root node has a single outgoing edge y (this is the final function value) which depends on variables $x_1, \cdots, x_n$, we can compute the derivative of y with respect to x_i by computing the derivative of y with respect to x_i through the root node, multiplied by the (recursively computed) derivatives of each incoming node to the root with respect to x_i. In this way, we can start at the top of the expression DAG and work backwards to build a sequence of dependencies, each one composition-level deep, of derivatives whose product is the final derivative of y with respect to x_i. For nodes representing arithmetic operations, such as addition, multiplication, or division, we simply expand the derivatives using the sum rule, product rule, or quotient rule, respectively. For nodes representing elementary functions such as the exponential function or trigonometric functions, we can reference a lookup table of pre-computed derivatives of elementary functions and use the chain rule to continue the recrusive steps.

Forward mode simply computes the earliest (in the DAG) derivatives first before multiplying it out, whereas backwards mode starts at the top and works backwards towards the beginning of the DAG, computing the product on the way.

3 Dual Numbers

The key to the computational efficiency and floating point accuracy of automatic differentiation lies in the way its implemented. The process is accompished not through explicit differentiation, but rather through an augmentation of the real numbers into an exterior algebra that imposes a new kind of arithmetic on the reals which allows us to, in some sense, map differentiation and the chain rule to the multiplication of elements of this new algebraic space. More specifically, we introduce a new kind of number called a dual number, which has the general form

$$x + \epsilon \cdot x'$$

where x and x' are real numbers, and ϵ is, informally, an infinitesimal, defined as the symbol obeying $\epsilon^2 = 0$. Immediately we have the following arithmetic on dual numbers:

$$(x + \epsilon \cdot x') + (y + \epsilon \cdot y') = (x + y) + \epsilon \cdot (x' + y')$$
$$(x + \epsilon \cdot x') \cdot (y + \epsilon \cdot y') = (x \cdot y) + \epsilon \cdot (x \cdot y' + x' \cdot y)$$
$$(x + \epsilon \cdot x')/(y + \epsilon \cdot y') = (x/y) + \epsilon \cdot (x \cdot y' - x' \cdot y)/y^2$$

This obeys precisely the relationships we'd expect from a derivative. This arithmetic allows us to pass dual numbers through polynomials P as $eP(x + \epsilon \cdot x') = P(x) + \epsilon \cdot x' P'(x)$ where P' is the derivative. From there, we can extend dual numbers to general smooth functions, including of course elementary functions:

$$\forall \textit{ smooth functions } f : f(x + \epsilon \cdot x') = f(x) + \epsilon \cdot f'(x)x'$$

Further, these properties carry over to multidimensional functions as well. The chain rule operations we described above in forward or backward mode now reduce to simple arithmetic operations of dual numbers. In particular, for a function f and point x_0 we can compute $f'(x_0)$ as $f(x_0 + \epsilon \cdot 1)$, which equals $f(x_0) + \epsilon \cdot f'(x_0)$. Hence, because the arithmetic of dual numbers is only a constant multiple f the time complexity of arithmetic on reals (or more accurately, their floating point representations), the computation of $f'(x_0)$, which only requires the computation of the function itself but with dual numbers instead of real numbers, is a constant multiple of the time complexity of computing f.