Literature Review on Automatic Differentiation in Machine Learning

Heng Wei

Introduction

The fields of machine learning and automatic differentiation (AD) have only recently converged. One of the main ways neural networks "learn" is by updating its numerous parameters with a procedure called Stochastic Gradient Descent. SGD requires the computation of gradients, hence the relevance of AD in machine learning. This review revisits the mathematical and technical foundations of AD, different ways to implement it in functional programming, and lastly explores some interesting areas of active research (policy gradients, randomized automatic differentiation, etc).

1 Background [1]

Before automatic differentiation, there were 3 other methods of computing the derivative in computer programs:

- Manually working out the derivative and coding them
- Numerical differentiation using finite difference approximations
- Symbolic differentiation using expression manipulation

Manual differentiation involves manually deriving and coding the derivative of functions. This method is impractical for complex models and is prone to human error. Numerical differentiation is inexact, and research has shown that even the most advanced numerical techniques face a tradeoff between complexity and approximation error. In deep learning, where n is often astronomically large, the best-case complexity of $O(n)$ for numerical methods makes them impractical. Symbolic differentiation, on the other hand, suffers from the issue of "expression swell." Additionally, both manual and symbolic differentiation require the function to be in a closed form, limiting the algorithmic control flow and expressiveness of the functions we aim to differentiate. These issues have significantly hindered research in machine learning until the discovery of automatic differentiation (AD) by the machine learning community.

2 Automatic Differentiation (AD) [1]

2.1 Overview

Mathematically, we are looking to compute the partial derivative of a function w.r.t. a certain parameter, by accumulating values of the partial derivatives at each step (computed via symbolic differentiation) to form the value of the final partial derivative we are looking for. The problem of "expression swell" is mitigated because the derivative of each sub-functions is evaluated and passed on. In other words, we do not output the expression of the derivative, but rather its exact value. By interleaving as much as possible our computations, the technique become computationally feasible even for mind-boggling big models AD has ideal asymptotic efficiency and only a small constant overhead. AD has ideal asymptotic efficiency and only a small constant overhead (AD has ideal asymptotic efficiency and only a small constant overhead).

Note: the sub-functions are often elementary functions with known derivatives (hence why we use symbolic differentiation). Furthermore we know that all numerical computations are ultimately composed of a finite set of elementary operations for which the derivative is known (Verma, 2000).

As a technical term, AD refers to a specific family of techniques to help us achieve this. In machine learning, we set our parameters to an initial set of values, and then compute the value of the function with a given input, before running AD.

2.2 Forward Mode AD

In forward mode AD, as the name suggests, we do a forward pass by feeding an input $\mathbf{x} = \mathbf{a}$, or in other words, we follow the computation graph starting from the input all the way to the output.

There are two things we need to compute and keep track of during the forward pass: the forward primal trace and its corresponding tangent/derivative trace. The values evaluated at each node during the forward pass have to be stored, this is the forward primal trace. The derivatives evaluated at each node during the forward pass have to be stored too (each node is an elementary operation for which the derivative is known), this is the corresponding tangent/derivative trace. Together, they form a tuple (v, v'), called a dual number.

Mathematically, we express it as $v + (v')\epsilon$, where ϵ is a nilpotent number ($\epsilon^2 = 0, \epsilon \neq 0$). By setting up a regime where $f(v+v') = f(v) + f(v')v'\epsilon$, with f being a node in our computation graph, we obtain the corresponding dual number at each node.

In practice, there are two ways to set up this regime. One requires taking the code of a certain function f, and generating a new code that handles the dual operation so that the value and derivative are simultaneously computed. Another way is through operator overloading.

Forward AD can be applied to functions $f : \mathbb{R}^n \to \mathbb{R}^m$. By feeding it an input $\mathbf{x} = \mathbf{a}$, we have $\mathbf{x}' = \mathbf{e_i}$ (we are looking for $\begin{bmatrix} \frac{\partial y_1}{\partial x_i} & \frac{\partial y_2}{\partial x_i} & \cdots & \frac{\partial y_m}{\partial x_i} \end{bmatrix}^T$), therefore one forward pass gives us the i-th column of the Jacobian of f (the partial derivatives of f w.r.t. the i-th component of $\mathbf{x}$). If $m = 1$, as is often the case in machine learning since f is the loss function we wish to minimize, then after one pass we would have $\frac{\partial y}{\partial x_i}$. The issue here is that we must compute n passes to obtain all our partial derivatives (∇f), and in machine learning, n is often astronomically big.

Note: generally, when $\mathbf{x}' = \mathbf{r}$, one forward pass computes efficiently the Jacobian-vector product (JVP), $J_f\mathbf{r}$ without explicitly constructing the Jacobian nor performing any matrix operations.

$$J_f\mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} \frac{\partial y_1}{\partial x_j} r_j \\ \vdots \\ \sum_{j=1}^{n} \frac{\partial y_m}{\partial x_j} r_j \end{bmatrix}$$

2.3 Reverse Mode AD

Reverse mode AD, also known as backpropagation, is particularly useful for neural networks and large-scale models. It remediates the issue of needing to compute n forward passes in forward mode AD. Instead of starting from the input and deriving w.r.t to a certain parameter (forward mode), we propagate derivatives from the output back to the input. We can then compute all partial derivatives with only one pass.

References

[1] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(153):1–43, 2018.