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]

Other than automatic differentiation, there are 3 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, the number of weights, 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 (reverse mode AD), the technique becomes computationally feasible even for mind-boggling big models (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).

2.2 Forward Mode AD

Let a model $f : \mathbb{R}^n \to \mathbb{R}^m$ be a composition of many elementary operations/nodes. In forward mode AD, we define a forward pass as feeding the model an input $\mathbf{x} = \mathbf{a}$, and then computing and storing the two things below as we follow the computation graph starting from the input all the way to the output:

1. The function evaluation at each node, v_i. This is called the forward primal trace. Each node is an elementary operation for which the derivative is known.

2. The derivative evaluation at each node, v_i'. This is called the corresponding tangent/derivative trace. The derivative evaluation can be done w.r.t. any x_i, the i-th component of $\mathbf{x}$, or any intermediary variables v_i.

Together, they form a list of tuples (v, v'), called a dual number.

Mathematically, we express a dual number as $v + (v')\epsilon$, where ϵ is a nilpotent number ($\epsilon^2 = 0, \epsilon \neq 0$). By setting up a regime where $h(v+v') = h(v)+h(v')v'\epsilon$, with h being any node in our computation graph, we obtain iteratively the corresponding dual number. In practice, we can set up this regime by taking the source code of a certain model E, and change it so that it handles the dual operation of simultaneously computing v and v'. Another way is through operator overloading.

In general, 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 $\mathbf{x}' = \mathbf{e_i}$, therefore one forward pass gives us the i-th column of the Jacobian of f:

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_i} & \frac{\partial y_2}{\partial x_i} & \cdots & \frac{\partial y_m}{\partial x_i} \end{bmatrix}^T$$

Even more generally, if we set $\mathbf{x}' = \mathbf{r}$, one forward pass computes efficiently the Jacobian-vector product (JVP),

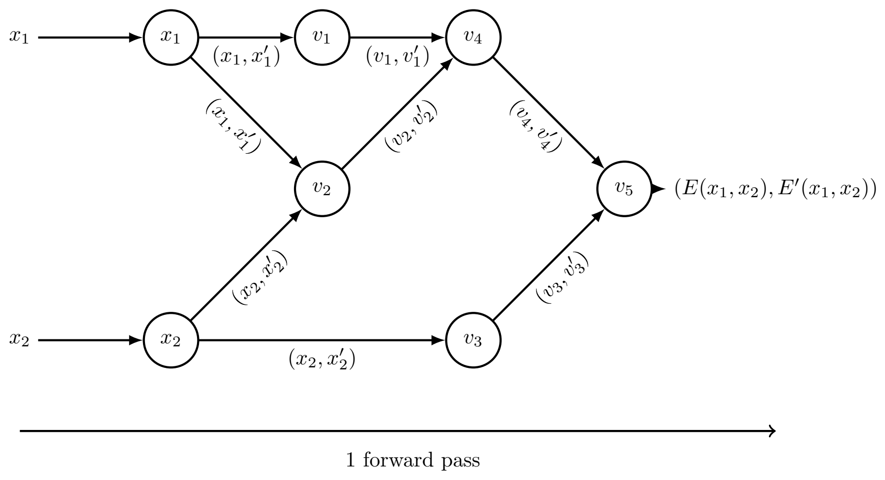
$$J_f \mathbf{r} \Big|_{\mathbf{x}=\mathbf{a}}$$

without explicitly constructing the Jacobian nor performing any matrix operations.

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

Important note: In the context of deep learning, after feeding the input, the entire neural network is defined as $E : \mathbb{R}^n \to \mathbb{R}$, where n is the number of intermediary variables or weights, not the dimension of the input. Therefore, after one pass we would have $\frac{\partial E}{\partial v_i}$. The issue here is that we must compute n passes to obtain all our partial derivatives ($\nabla E = J_E = \begin{bmatrix} \frac{\partial E}{\partial v_1} & \frac{\partial E}{\partial v_2} & \cdots & \frac{\partial E}{\partial v_n} \end{bmatrix}^T$). In deep learning, n is often astronomically big.

For example, a forward pass in a neural network with 4 intermediary variables, no matter which v_i we are deriving w.r.t., would look like this:



Note: for any node v_i that serves as input to a node v_j, we consider $\frac{\partial v_i}{\partial v_j} = 0$, and not $\frac{\partial v_i}{\partial v_j} = \frac{\partial v_j}{\partial v_i}^{-1}$, because updating a node in our graph does not affect the nodes that serves as its input. In other words, if $v_j = h_i(v_i)$, we do not interpret $v_i = h_i^{-1}(v_j)$.

2.3 Reverse Mode AD

Reverse mode AD, also known as backpropagation, is particularly useful for neural networks and large-scale models. A pass in reverse mode is composed of two stages. The first stage is a forward pass that builds only the forward primal trace, i.e. populates all intermediary variables v_i. The second stage is a backward pass, starting from the output and going back towards the input, that builds the reverse adjoint trace. The ajoint is defined as the partial derivative of the final output y_i by an intermediary variable:

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$

In general, reverse mode AD can be applied to functions $f : \mathbb{R}^n \to \mathbb{R}^m$. One reverse pass (composed of a forward and backward pass) gives us i-th row of the Jacobian of f:

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \end{bmatrix}$$

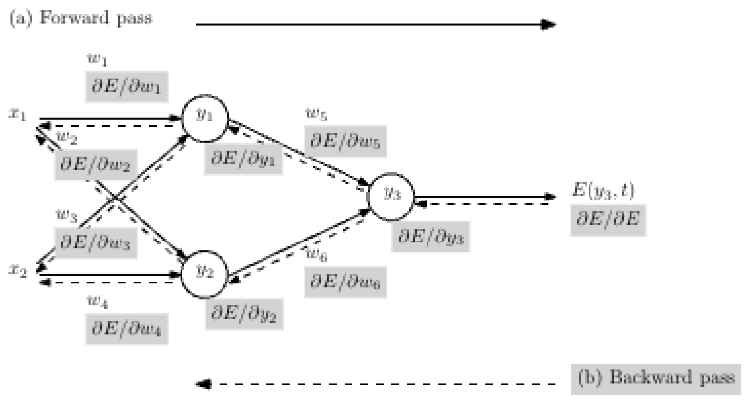
Even more generally, if we initialize the backward stage with $\bar{\mathbf{y}} = \mathbf{r}$, one reverse pass computes efficiently the vector-Jacobian product (VJP),

$$\mathbf{r}^T J_f \Big|_{\mathbf{x}=\mathbf{a}}$$

without explicitly constructing the Jacobian nor performing any matrix operations.

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

Important note: In the context of deep learning, contrary to forward mode that requires n forward passes to compute the gradient (assuming $E : \mathbb{R}^n \to \mathbb{R}$), reverse mode does it in 1 reverse pass. After the forward pass the entire neural network is defined as $E : \mathbb{R}^n \to \mathbb{R}$, where n is the number of weights. To obtain all our partial derivatives ($\nabla E = J_E = \begin{bmatrix} \frac{\partial E}{\partial w_1} & \frac{\partial E}{\partial w_2} & \cdots & \frac{\partial E}{\partial w_n} \end{bmatrix}^T$), we start the backward pass with $\frac{\partial E}{\partial E} = 1$. With the chain rule of total derivatives, we store our partial derivatives and accumulate them. For example, a reverse pass in a neural network with 6 weights would look like this:


(a) Forward pass
w_1
$\partial E/\partial w_1$
x_1
y_1
w_2
$\partial E/\partial w_2$
w_5
$\partial E/\partial w_5$
$\partial E/\partial y_1$
y_3
$E(y_3, t)$
$\partial E/\partial E$
$\partial E/\partial y_3$
w_3
$\partial E/\partial w_3$
x_2
y_2
w_6
$\partial E/\partial w_6$
w_4
$\partial E/\partial w_4$
$\partial E/\partial y_2$
(b) Backward 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.