Derek Lee Deep Learning Fall 2020

Professor Curro Quiz #1

1. Compare and contrast symbolic differentiation, numeric differentiation, and automatic differentiation.

Numeric differentiation involves using finite difference approximations to calculate a derivative. It is simple to implement, being based on the limit definition of a derivative, but "can be highly inaccurate due to round-off and truncation errors (Jerrell, 1997)". During the demonstration during the lecture, you (Prof. Curro) showed a finite difference approximation to calculate the derivative. Once you used a small enough number (1e-16), the derivative became 0, rather than 2. Additionally, it requires O(n) evaluations for a gradient in n dimensions. In deep learning models, n can be on the order of millions or billions, which makes this method unfeasible. Notably, the approximation errors from using this method "would be tolerated… thanks to the well-documented error resiliency of neural network architectures."

Symbolic differentiation relies on expression manipulation in computer algebra systems. It does not have the same problem as numeric differentiation. Instead, it suffers from memory constraints, especially prominent in complicated models which require many applications of the chain rule. It can get "exponentially larger than the expression whose derivative [it] represent[s]." In addition, it can result in "complex and cryptic expressions". It depends on models being defined as closed-form expressions, which leads to unnatural programming. There are some optimizations available, such as simplifying computations by "storing only the values of intermediate sub-expressions in memory" and interleaving "as much as possible the differentiation and simplification steps."

Automatic differentiation addresses the weaknesses in both above techniques. It is partly symbolic and partly numerical, using symbolic rules of differentiation, but computing numerical values rather than derivative expressions. It computes derivatives "through accumulation of values during code execution to generate numerical derivative evaluations rather than derivative expressions." This means it can evaluate derivatives at machine precision while avoiding having

to arrange code in closed-form expressions, so it does not require unnatural programming. On top of this, it only has a small constant factor of overhead.

2. Compare and contrast forward-mode automatic differentiation with reverse-mode automatic differentiation.

Forward-mode automatic differentiation is very efficient for functions that map from n to m, where n << m. When n=1, it can be computed with a single forward pass. When m=1, it needs n evaluations. It starts with associating each intermediate variable vi with a partial derivative of the intermediate variable with respect to each input variable. Then, the chain rule is applied to each elementary operation in the forward primal trace, which generates a corresponding tangent trace. Evaluating the primals "in lockstep with their corresponding tangents" results in the derivative in the final variable. This generalizes well to computing the Jacobian of a function. Only one input variable derivative is set equal to one for each iteration. The other derivatives are set to zero. This computes the full Jacobian in n evaluations. This can also be used to compute Jacobian-vector products by initializing derivative of x = r. This results in the Jacobian-vector product in one forward pass.

Reverse-mode automatic differentiation is very efficient for functions that map from n to m, where m << n. When m=1, it can be computed with a single application. When n=1, it needs m applications. "Because machine learning practice principally involves the gradient of a scalar-valued object with respect to a large number of parameters, this establishes the reverse mode, as opposed to the forward mode, as the mainstay technique in the form of the backpropagation algorithm." It is a generalized version of the backpropagation algorithm. It associates each intermediate variable with an adjoint, representing the partial derivative of each output variable with respect to that intermediate variable. Each adjoint represents how sensitive the output variable is to changes in the intermediate variable. It uses a two-phase process; First, it runs the forward primal trace, like forward-mode automatic differentiation. After that, it propagates adjoints in reverse, from outputs to inputs. Similar to the computation of Jacobian-vector products in forward-mode, backward-mode can compute the transposed Jacobian-vector product by initializing the second phase with adjoint of y = r.