ECE472 – Quiz 1

Jonathan Lam

September 9, 2020

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

   Symbolic differentiation This provides a (usually closed-form) expression for the derivative of an expression w.r.t. (some or all of) its input variables. This has the benefits of being mathematically exact and easy to interpret; having the closed form expression for a derivative may be useful, e.g., for finding extrema (zeros of the derivative expression) analytically. However, this suffers from expression swell (since repeated chain rule on complex expressions can quickly make the expression grow very long), which can take a lot of memory and lead to many repeated calculations. This also suffers from manipulating code to generate the derivative expression, which may be unwieldy.

   Numeric differentiation This approximates the derivative using a difference quotient. While this is faster than symbolic differentiation for complex expressions, it is very prone to error. The error not only grows with the length of the calculation, but round-off and truncation errors occur due to the inherent nature of the derivative being defined as a limit where the denominator approaches zero, and of the limited size of numeric types on computers.

   Automatic differentiation This is the preferred way of doing differentiation of an expression with respect to its inputs and outputs, since it provides exact values like symbolic differentiation (within the limits of numeric types) and provides only constant-time overhead for each input variable (like numeric differentiation). This works by performing the same calculations as symbolic differentiation (and therefore producing an identical result), but not storing or calculating the entire derivative expression. Rather, it only stores the values of subexpressions and their derivatives (at a small memory overhead), which can be used by the chain rule to calculate the derivatives of larger subexpressions (in forward accumulation mode; the chain rule is applied in a slightly different way in reverse accumulation mode).

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

   Any particular expression s has a series of constants and variables (the most primitive subexpressions) and a hierarchical tree of subexpressions. Except for the entire expression s, each subexpression s_1 is combined with another subexpression s_2 to form another subexpression s_3 (in the case of a binary expression, but it is not hard to see how this would work with operations with different arities). To calculate the value of the subexpression s_3, we have to know (i.e., calculate and store) the values of s_1 and s_2 before hand; so we always evaluate expressions from the inside-out, calculating more primitive subexpressions before being able to calculate the value of the expressions that depend on them.

   Forward-mode autodiff works the same way. Assume there is a variable x, and we want to find $\frac{\partial s_3}{\partial x}$. Instead of only requiring the *values* of s_1 and s_2 before calculating this partial derivative, we also have to know $\frac{\partial s_1}{\partial x}$, $\frac{\partial s_2}{\partial x}$, $\frac{\partial s_3}{\partial s_1}$, and $\frac{\partial s_3}{\partial s_2}$, since the chain rule states:

$$\frac{\partial s_3}{\partial x} = \frac{\partial s_3}{\partial s_1}\frac{\partial s_1}{\partial x} + \frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial x}$$

   The latter two depends on the operation being performed on s_1 and s_2 to form s_3, and the former two have already been calculated by performing this kind of calculation recursively from the lower expressions upwards.

   This is straightforward because it is a direct application of the chain rule, but it involves calculating and recording the partial derivatives of every intermediate value w.r.t. each input variable. Reverse-mode differentiation is different in that we can aggregate the values in the opposite direction: the same chain rule equation above can be accumulated in reverse. For example, imagine that there is a subexpression s_4, which is used directly in subexpressions s_5, s_6, and s_7. This gives the following equation (also by chain rule), which looks similar to the above equation. (It's still chain rule, but from the output variable's perspective.)

$$\frac{\partial y}{\partial s_4} = \frac{\partial y}{\partial s_5}\frac{\partial s_5}{\partial s_4} + \frac{\partial y}{\partial s_6}\frac{\partial s_6}{\partial s_4} + \frac{\partial y}{s_7}\frac{\partial s_7}{\partial s_4}$$

   Algorithmically, we start from calculating the partial derivatives of the subexpressions that y is directly dependent on through this relation. Then, we can move "backwards": in this case, we calculate $\frac{\partial y}{\partial s_4}$ based on the partial derivatives of its dependencies, i.e., $\frac{\partial y}{\partial s_5}$, $\frac{\partial y}{\partial s_6}$, and $\frac{\partial y}{\partial s_7}$.

   Why do this? While forward accumulation builds up partial derivatives for each subexpression w.r.t. each input variable, reverse accumulation builds up partial derivatives for each subexpression w.r.t. each output variable. Since we usually have many more features than output variables, the former is much more expensive and calculates a lot of values we don't need, since we don't care about the partial derivatives of each of the subexpressions w.r.t. input variables.