Seminar
Lecture #1:Introduction to Automatic Differentiation

David Juedes

Ohio University Russ College of Engineering and Technology

Feb. 25th, 2020

Differentiation? Is it interesting?

Like many things in computer science, relatively simple concepts sometimes have interesting computational properties, e.g., determining whether a boolean formula is satisfied on a given setting of the inputs is easy, but finding such a setting is NP-complete.
Differentiation is also one of those concepts that has that permeates science and engineering, but is not as easy to compute efficiently as you might expect.

Why this is interesting?

We use derivatives to help solve all sorts of optimization problems.

- Neural Networks
- Robotics
- Protein Folding
- Etc.

Some of these computations can be expensive, and the derivative calculations themselves can be a significant component of the optimization calculation. Hence, if we can calculate derivatives more quickly, we win.
(These notes are partially based on a set of notes compiled by one of my graduate students, Patrick Hartmann, in 1997.)

Review from Calculus I, II, and III

Before I talk about *automatic differentiation*, let me give a quick overview of the relevant material from Calculus I (derivative), Calculus II (integrals), and Calculus III (vector calculus).

Give a function f on a single variable x, the derivative of f at x is defined to be

$$f'(x) = \lim_{\delta \longrightarrow 0} \frac{f(x+\delta) - f(x)}{\delta}.$$

We can think of the derivative as the instantaeous slope of the function at x.

Now, it's possible that this derivative does not exist.

Example #1

$$f(x) = x^2$$

$$\begin{aligned}
\lim_{\delta \longrightarrow 0} \frac{f(x+\delta) - f(x)}{\delta} &= \lim_{\delta \longrightarrow 0} \frac{x^2 + 2\delta x + \delta^2 - x^2}{\delta} \\
&= \lim_{\delta \longrightarrow 0} 2x + \delta \\
&= 2x
\end{aligned}$$

We can use the definition of differentiation to build the differentiation rules for a variety of standard functions. These are well-known.

Example #2

$$f(x) = cx$$

$$\begin{aligned}
\lim_{\delta \longrightarrow 0} \frac{f(x+\delta) - f(x)}{\delta} &= \lim_{\delta \longrightarrow 0} \frac{cx + c\delta - cx}{\delta} \\
&= \lim_{\delta \longrightarrow 0} c \\
&= c
\end{aligned}$$

Vector Calculus

If f is a multivariate function (e.g., f is a function of two variables x, and y), then f doesn't have a single derivative — it has a derivative in each direction given by its inputs. These are *partial* derivatives, e.g., if $z = f(x, y)$, then

$$\frac{\partial z}{\partial x} = \lim_{\delta \longrightarrow 0} \frac{f(x+\delta, y) - f(x, y)}{\delta}$$

and

$$\frac{\partial z}{\partial y} = \lim_{\delta \longrightarrow 0} \frac{f(x, y+\delta) - f(x, y)}{\delta}$$

Examples

Assume $x = y * z$.

$$\frac{\partial x}{\partial y} = 1 * z$$

$$\frac{\partial x}{\partial z} = y * 1$$

(Both are special cases of example #2. We can think of y or z as a constant in both cases.)

Basic Notation

- Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a composite differentiable function. (Composed of a sequence of elementary functions whose derivative properties are well-known, e.g., addition, multiplication, integrals.)
- We generally assume that f is represented by a program (algorithm).
- If $f(\langle x_1, \ldots, x_m \rangle) = \langle y_1, \ldots, y_n \rangle$ then the *Jacobian* of f is the $m \times n$ matrix of first-order partial derivatives of f,

$$J_f = \begin{bmatrix} & \frac{\partial y_j}{\partial x_i} & \\ & & \end{bmatrix}_{m \times n}$$

- If $f : \mathbb{R}^n \to \mathbb{R}$ and $f(\langle x_1, \ldots, x_n \rangle) = y$ then the *gradient* of f is the m element vector

$$\triangle_f = \begin{bmatrix} \frac{\partial y}{\partial x_i} \end{bmatrix}.$$

Example

Consider the following function $f(\vec{x}) = \prod_{i=1}^{n} x_i$ computed by the following function.

```
double f(vector<double> &x) {
  prod = 1.0;
  for (int i=0;i<x.size();i++) {
     prod*=x[i];
  }
  return prod;
}
```

Computing this takes $O(n)$ steps.

Symbolic Differentiation

If we look gradient of f symbolically, we get

$$\Delta_f = \begin{bmatrix} \prod_{i=2}^{n} x_i \\ x_1 * \prod_{i=3}^{n} \\ \cdots \\ \prod_{i=1}^{j-1} x_i * \prod_{i=j+1}^{n} x_i \\ \cdots \\ \prod_{i=1}^{n-1} x_i \end{bmatrix}$$

Computing this directly takes $O(n^2)$ steps.

Divided Differencing

Another approach to calculating gradients is to approximate the partial derivations by applying the definitions for sufficiently small values of δ

```
z = f(x);
xp=x;
for (int i=0;i<x.size();i++) {
    xp[i] = x[i] + delta;
    grad[i] = (f(xp) - z)/delta;
    xp[i] = x[i]; //restore
}
```

For divided differencing, picking the right δ may be hard.
In this case, divided differencing takes $O(n^2)$ steps to calculate the gradient.
We'll show you how to calculate the gradient *exactly* (no approximation like above) in $O(n)$ steps using automatic differentiation.

Basics of AD

Recall the chain rule....

Lemma (Chain Rule)

If $y = f(u)$, $u = g(x)$, *and* $\frac{\partial y}{\partial u}$ *and* $\frac{\partial u}{\partial x}$ *are known, then*

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}.$$

Automatic Differentiation relies on a simple extension of the chain rule.

Extension of Chain Rule

Lemma (Extension of Chain Rule)

Let $f_1 : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$, $f_2 : \mathbb{R}^{n_2} \to \mathbb{R}^{n_3}$, *and* $f_3 = f_2 \circ f_1$.
If J_{f_1} *and* J_{f_2} *are known, then*

$$J_{f_3} = J_{f_1} \times J_{f_2}.$$

AD relies on the fact that the Jacobians for the elementary functions are simple and very sparse.

AD

- To start, consider an expression such as

$$x = ab + b^2.$$

- Using a parser, a tree structure can be created from this expression.
- Traversing this parse tree (in a post-order manner), a *code list* can be made of binary operations that compute the result, x.

$$\begin{aligned} y &= a \times b && (1) \\ z &= b \times b && (2) \\ x &= y + z && (3) \end{aligned}$$

- Each operation can be differentiated locally.
- We can apply the chain rule to compute $\frac{\partial x}{\partial a}$ and $\frac{\partial x}{\partial b}$.

OHIO UNIVERSITY

AD

For simplicity, we consider each elementary operation as function that transforms the programs variables to the new values of those variables. In our example, the three elementary operations is viewed as functions that map $\mathbb{R}^5$ to $\mathbb{R}^5$.
In our example, we get the following Jacobians for our elementary functions.

$$J_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & b & 0 & 1 & 0 \\ 0 & a & 0 & 0 & 1 \end{bmatrix} \begin{matrix} \leftarrow \partial x \\ \leftarrow \partial y \\ \leftarrow \partial z \\ \leftarrow \partial a \\ \leftarrow \partial b \end{matrix}$$

$$J_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2b & 0 & 1 \end{bmatrix} \leftarrow \frac{\partial z}{\partial b} = 2b$$

$$J_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} \leftarrow \frac{\partial x}{\partial y} = 1 \\ \leftarrow \frac{\partial x}{\partial z} = 1 \end{matrix}$$

AD

Multiplying these matrices together results in the following.

$$J_1 \times J_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & b & 0 & 1 & 0 \\ 0 & a & 2b & 0 & 1 \end{bmatrix}$$

$$J_1 \times J_2 \times J_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & b & 0 & 1 & 0 \\ 0 & a & 2b & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ b & b & 0 & 1 & 0 \\ a+2b & a & 2b & 0 & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ \leftarrow \frac{\partial x}{\partial a} \\ \leftarrow \frac{\partial x}{\partial b} \end{matrix}$$

AD

Often, calculating the gradient of a function is of interest to us. In our approach, this is one column of the final Jacobian.
Assume that our computation is formed from n elementary operations, let m be the number of variables in the computation, and let J be the final $m \times m$ Jacobian.
The Forward Mode:
If $\vec{x}$ is an m-element vector,then we can compute $\vec{x}^T J$ in the following straightforward manner using only $O(n)$ operations.

$$(\vec{x})^T \times J = ((((\vec{x})^T \times J_{f_1})^T \times J_{f_2})^T \ldots)^T \times J_{f_n})$$

To compute the gradient with this approach takes $O(m \times n)$ steps. We can do better!

The Reverse Mode

Let $\vec{x}$ be an m-element vector. We can compute $J \times \vec{x}$ in $O(n)$ steps. To see this, notice that

$$J = J_{f_1} \times J_{f_2} \times \ldots \times J_{f_n},$$

and hence that

$$J \times \vec{x} = (J_{f_1} \times (J_{f_2} \times (\ldots \times (J_{f_n} \times \vec{x}))))$$

Since each J_{f_i} is simple, we can perform each matrix-vector multiplication with only $O(1)$ steps.
To compute the gradient, simply let $\vec{x}$ be a unit vector.

Example

In our previous example, let

$$\vec{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Cont'd

Then, $J \times \vec{x} =$

$$J \times \vec{x} = J_1 \times J_2 \times \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= J_1 \times \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2b & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Cont'd

$$= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & b & 0 & 1 & 0 \\ 0 & a & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 2b \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ 1 \\ 1 \\ b \\ a+2b \end{bmatrix}$$

When multiplying an elementary Jacobian and a vector $\vec{x}$, the only values of $\vec{x}$ that change are those that correspond to the arguments of the function. Consider the following examples.

Example

Example: $f = x_i = x_j + x_k$
In this case,

$$J_f \times \vec{\bar{x}} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{x}_i \\ \bar{x}_j \\ \bar{x}_k \end{bmatrix}$$

$$= \begin{bmatrix} \bar{x}_i \\ \bar{x}_i + \bar{x}_j \\ \bar{x}_i + \bar{x}_k \end{bmatrix}$$

Example

Example: $f = x_i = x_j * x_k$

In this case,

$$J_f \times \vec{\dot{x}} = \begin{bmatrix} 1 & 0 & 0 \\ x_k & 1 & 0 \\ x_j & 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{x}_i \\ \bar{x}_j \\ \bar{x}_k \end{bmatrix}$$

$$= \begin{bmatrix} \bar{x}_i \\ \bar{x}_i * x_k + \bar{x}_j \\ \bar{x}_i * x_j + \bar{x}_k \end{bmatrix}$$

Cont'd

This gives the following simple rules.
For Addition:

$$\begin{aligned} \bar{x_j} &+= \bar{x_i} \\ \bar{x_k} &+= \bar{x_i} \end{aligned}$$

For Multiplication:

$$\begin{aligned} \bar{x_j} &+= \bar{x_i} * x_k \\ \bar{x_k} &+= \bar{x_i} * x_j \end{aligned}$$

Adjoint Equations

The previous slide gives the *adjoint equations* for addition and multiplication.
To implement that adjoint equations, we can keep a separate variable for each variable in the program. Applying the adjoint equations in reverse order from the original computation computes all the partial derivatives (if you set the adjoint values appropriately to begin with).
Notice, if a value gets overwritten during the computation, we replace the adjoint value instead of adding to its current value.

Example – Product

```
#include <iostream>
#include <vector>
using namespace std;
double prod(vector<double> &x) {
  double prod = 1.0;
  for (int i=0;i<x.size();i++) {
    prod = prod * x[i];
  }
  return prod;
}
int main() {
  vector<double> x;
  x.push_back(2.0);
  x.push_back(3.0);
  x.push_back(4.5);
  x.push_back(1.5);
  x.push_back(1.3);
  cout << prod(x) << endl;
}
```

Adjoint Code

```
// Produces function and all partial derivatives in O(n) steps.
//
#include <iostream>
#include <vector>
#include <stack>
#include <cassert>
using namespace std;
stack<double> rev_stack;

double adj_prod(vector<double> &x, vector<double> &adj_x) {
  double prod = 1.0;
  for (int i=0;i<x.size();i++) {
    rev_stack.push(prod);
    rev_stack.push(x[i]);
    prod = prod * x[i];
  }
```

Adjoint Code

```
  // Run the code in reverse!
  int j = x.size()-1;
  double prod_adj = 1.0;
  adj_x.resize(x.size(),0.0);
  while (!rev_stack.empty()) {
    double x_val;
    double prod_val;
    x_val=rev_stack.top();
    rev_stack.pop();
    prod_val = rev_stack.top();
    rev_stack.pop();
    // Apply adjoint equations
    adj_x[j]+=prod_adj*prod_val;
    prod_adj=prod_adj*x_val;
    j--;
  }
  assert(j==-1);
  return prod;
}
```

Adjoint Code

```
int main() {
  vector<double> x;
  vector<double> grad_x;
  x.push_back(2.0);
  x.push_back(3.0);
  x.push_back(4.5);
  x.push_back(1.5);
  x.push_back(1.3);

  cout << adj_prod(x,grad_x) << endl;

  cout << "Gradient" << endl;
  for (int i=0;i<grad_x.size();i++) {
    cout << i << " " << grad_x[i] << endl;
  }
}
```

Output

Notice that the vector x is $[2, 3, 4.5, 1.5, 1.3]$, and that the partial derivative of the product with respect to each x_i is $prod/x_i$.

```
52.65
Gradient
0 26.325
1 17.55
2 11.7
3 35.1
4 40.5
```

This code took $O(n)$ steps with $O(n)$ additional memory.
Notice, my code used no division!

Complexities

Discussion about complexities of differentiation.

1. Storing the intermediate values for long computations.
2. Computer programs that compute higher order functions
   - Example: Integrals
3. Libraries — e.g., LINPACK
4. Computing the full Jacobian
5. Higher order derivatives (Hessians, etc.)
6. Computing large/sparse Jacobians efficiently (graph coloring).
7. Compiler techniques
8. ADOL-C.

Some References

These techniques have been known for many years. The reverse mode has been known since at least 1970 (see Linnainmaa below). Here are some classic papers/books.

1. Linnainmaa, Seppo (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 6-7.
2. Rall, Louis B. (1981). Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science. 120. Springer. ISBN 978-3-540-10861-0.
3. Speelpenning, Bert (1980). Compiling Fast Partial Derivatives of Functions Given by Algorithms. Ph.D. Dissertation, University of Illinois, Urbana Champaign