Clad - Clang plugin for Automatic Differentiation

Jack Qiu
Google Summer of Code 2019
Mentors: Aleksandr Efremov, Vassil Vassilev, Oksana Shadura
Automatic Differentiation - Forward Mode

- Differentiation is fixed w.r.t to a independent variable
- Breaks a function up into a list of sub-expressions/sequence of elementary operations
- Computes the derivative of each sub-expression recursively
- Implemented in Clad through clad::differentiate

\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \left( \frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x} \right) = \frac{\partial y}{\partial w_{n-1}} \left( \frac{\partial w_{n-1}}{\partial w_{n-2}} \left( \frac{\partial w_{n-2}}{\partial w_{n-3}} \frac{\partial w_{n-3}}{\partial x} \right) \right) = \ldots
\]
Automatic Differentiation - Reverse Mode

- Differentiation is fixed w.r.t the dependent variable
- We break function into sub-expressions, apply chain rule starting from the dependent variable
- Very effective for large no. of independent variables, but requires significant computation memory
- Implemented in Clad through clad::gradient

\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1} \cdot \frac{\partial w_1}{\partial x} = \left( \frac{\partial y}{\partial w_2} \frac{\partial w_2}{\partial w_1} \frac{\partial w_1}{\partial x} \right) = \left( \frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \ldots
\]
What CLAD is

- Uses Clang compiler to perform source-code transformation
- Traverses through Clang AST with clang::StmtVisitor and builds a derivative function
- Can perform both forward and reverse mode AD

```cpp
double f(double x) {
    return x * x;
}
```
What can be differentiated

- Built-in C/C++ scalar types (e.g. double, float, int)
- Built-in C input arrays
- Functions that have an arbitrary number of inputs
- Functions that return a single value
- Loops
- Conditionals
double f_cubed_add1(double a, double b) {
    return a * a * a + b * b * b;
}

double f_cubed_add1_darg0(double a, double b) {
    double _d_a = 1;
    double _d_b = 0;
    double _t0 = a * a;
    double _t1 = b * b;
    return (_d_a * a + a * _d_a) * a + _t0 * _d_a + (_d_b * b + b * _d_b) * b + _t1 * _d_b;
}
```cpp
def double f_cubed_add1(double a, double b) {
    return a * a * a + b * b * b;
}
```

```cpp
void f_cubed_add1_grad(double a, double b, double *result) {
    double t0;
    double t1;
    double t2;
    double t3;
    double t4;
    double t5;
    double t6;
    double t7;
    t2 = a;
    t1 = a;
    t3 = t2 * t1;
    t0 = a;
    t6 = b;
    t5 = b;
    t7 = t6 * t5;
    t4 = b;
    double f_cubed_add1_return = t3 * t0 + t7 * t4;
    goto _label0;
    _label0:
    {
        double r0 = 1 * t0;
        double r1 = r0 * t1;
        result[0UL] += r1;
        double r2 = t2 * r0;
        result[0UL] += r2;
        double r3 = t3 * 1;
        result[0UL] += r3;
        double r4 = 1 * t4;
        double r5 = r4 * t5;
        result[1UL] += r5;
        double r6 = t6 * r4;
        result[1UL] += r6;
        double r7 = t7 * 1;
        result[1UL] += r7;
    }
}
```
Hessians

- Square $n \times n$ matrix containing all second order partial derivatives w.r.t to all inputs
- Useful for optimisation problems and as a second derivative test
Hessians - How it is implemented

- Generated through using forward mode AD, then reverse mode AD
- Iteratively calculates each column of the Hessian at a time, which is encapsulated within a second-order partial derivative function
- Combines all of these helper functions that correspond to columns of a Hessian into a single Hessian function
- Encapsulated in Clad API through clad::hessian
double f_cubed_add1(double a, double b) {
    return a * a * a + b * b * b;
}

auto func = clad::hessian(f_cubed_add1);
func.dump();

void f_cubed_add1_hessian(double a, double b, double *hessianMatrix) {
    f_cubed_add1_darg0_grad(a, b, &hessianMatrix[0UL]);
    f_cubed_add1_darg1_grad(a, b, &hessianMatrix[2UL]);
}
Hessians - Demo

double result[4];
func.execute(1.0, 2.0, result);

[6.0, 0.0, 0.0, 12.0]

\[
\begin{pmatrix}
6.0 & 0.0 \\
0.0 & 12.0
\end{pmatrix}
\]
Future Work

- Hessians
  - Finding a way to calculate the determinant
  - Resolving the 1-dimension array issue to allow for 2d array input and output
  - Benchmarking row-by-row approach

- General
  - Jacobians
  - Finding a way to compose forward and reverse mode together, i.e. 
    `clad::differentiate(clad::gradient(f))`
For more information, visit:
https://github.com/vgvassilev/clad