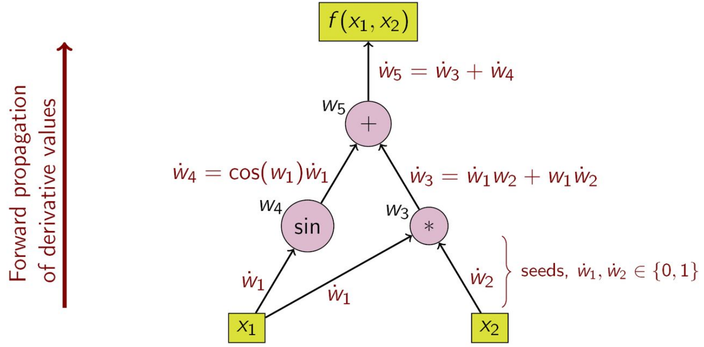
Clad - Clang plugin for Automatic Differentiation

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

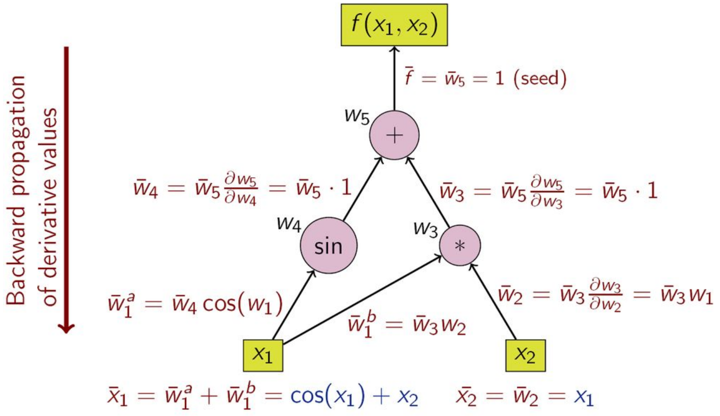
$$
\begin{aligned}
z &= f(x_1, x_2) \\
&= x_1 x_2 + \sin x_1 \\
&= w_1 w_2 + \sin w_1 \\
&= w_3 + w_4 \\
&= w_5
\end{aligned}
$$



$$
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x} \right) = \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \left(\frac{\partial w_{n-2}}{\partial w_{n-3}} \frac{\partial w_{n-3}}{\partial x} \right) \right) = \cdots
$$

Automatic Differentiation - Reverse Mode

- Differentiation is fixed w.r.t to 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



$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \cdots$$

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

```
double f(double x) {
     return x * x;
}
```

AST

```
TranslationUnitDecl
`-FunctionDecl <line:1:1, line:3:1> line:1:8 f 'double (double)'
|-ParmVarDecl <col:10, col:17> col:17 used x 'double'
`-CompoundStmt <col:20, line:3:1>
`-ReturnStmt <line:2:2, col:13>
`-BinaryOperator <col:9, col:13> 'double' '*'
|-ImplicitCastExpr <col:9> 'double' <LValueToRValue>
| `-DeclRefExpr <col:9> 'double' lvalue ParmVar 0x55be6537ee20 'x'
'double'
`-ImplicitCastExpr <col:13> 'double' <LValueToRValue>
`-DeclRefExpr <col:13> 'double' lvalue ParmVar 0x55be6537ee20 'x'
'double'
```

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

clad::differentiate

```
double f_cubed_add1(double a, double b) {
 return a * a * a + b * b * b;
}
```

clad::differentiate

```
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;
}
```

clad::gradient

```
double f_cubed_add1(double a,
double b) {
 return a * a * a + b * b * b;
}
```


```
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 x n matrix containing all second order partial derivatives w.r.t to all inputs
- Useful for optimisation problems and as a second derivative test

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\[2.2ex] \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\[2.2ex] \vdots & \vdots & \ddots & \vdots \\[2.2ex] \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

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

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Hessians - Demo

```
double f_cubed_add1(double a, double b) {
 return a * a * a + b * b * b;
}
```

```
auto func = clad::hessian(f_cubed_add_1);
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]
```

$$\begin{bmatrix} 6.0 & 0.0 \\ 0.0 & 12.0 \end{bmatrix}$$

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