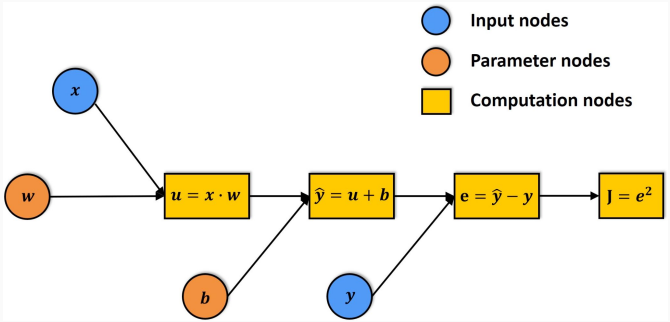
Vectorized forward mode AD in clad

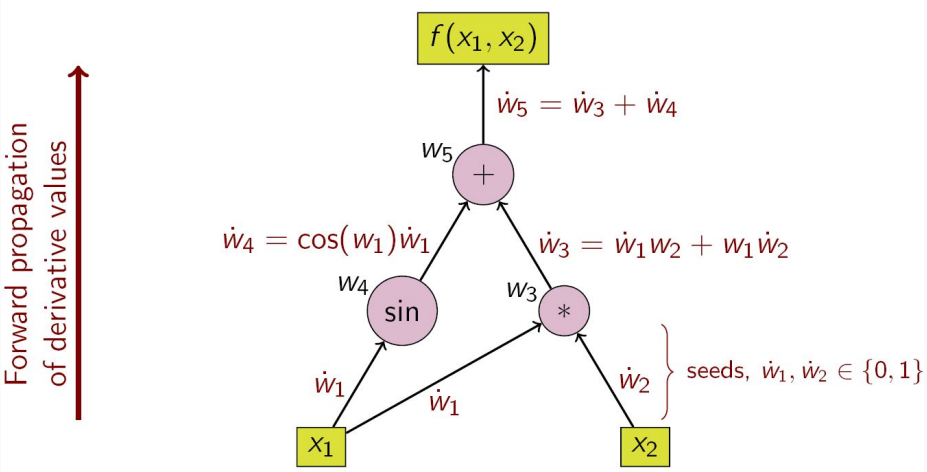
By Vaibhav Thakkar

Basics of Automatic Differentiation (AD)

- Aims to produce a procedure for computing the derivative of any general computational program.
- It does so by breaking down the program into a **computation graph** of primitive operations (+, -, * , /) and some primitive functions (sin, cos, exp …).

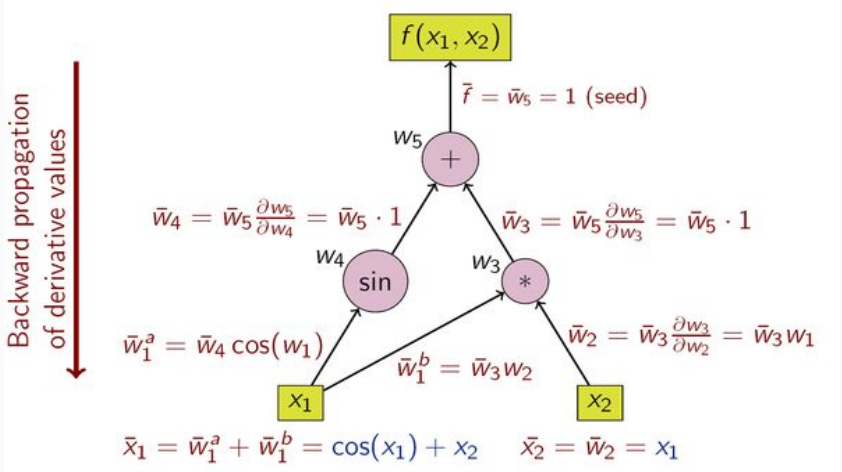


Forward mode vs Reverse mode AD



Forward mode AD

preferred when derivative w.r.t single input



Reverse mode AD

preferred when single output, multiple inputs gradient computations

Clad

A clang plugin to generate the code for derivative of a function at ***compile time***.

Source code transformation - analyzes the AST of the input program and generates AST for the derivative program.

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

clad::differentiate

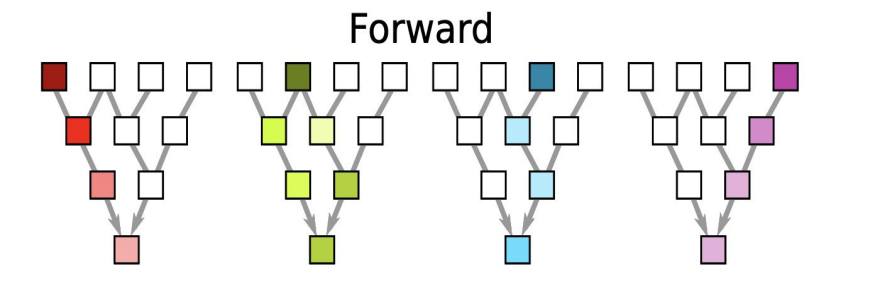
```
double f_cubed_add1_darg0(double a, double b) {
    double _d_a = 1;
    double _d_b = 0;
    double _t0 = a * a;
    double _t1 = b * b;
    return (_d_a * a + a * _d_a) * a + _t0 * _d_a + (_d_b *
b + b * _d_b) * b + _t1 * _d_b;
}
```

Vectorized forward mode

Problem

For computing gradient of a function with *n*-dimensional input - forward mode requires *n* forward passes, 1 for each input.

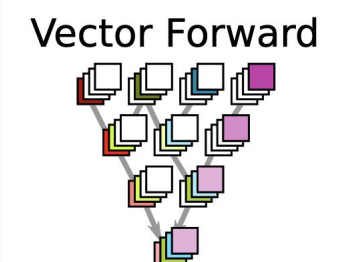
Can we instead compute the complete gradient in one pass?



Proposed Solution

Instead of accumulating a single scalar value of derivative with respect to a particular node - maintain a gradient vector at each node.

Initialised by a 1-hot vector for each input node



Why is it better than reverse mode AD?

Vectorized forward mode can effectively utilize parallelization and vectorization capabilities of modern CPUs and GPUs.

Reverse mode has large memory requirements as requires storing values of all intermediate nodes during forward evaluation.

Easy to implement and less error prone compared to reverse mode (especially for source transformation approach).

Updated clad interface

```
double fn(double x, double y) {
  double t = someComputationalIntensiveFn(); // should be computed only once
                                             // in the derived function.
    double res = 2 * t * x + 3 * t * x * y;
    return t;
  }

  int main() {
    auto d_fn = clad::differentiate(fn, "arr");
    double d_x = 0, d_y = 0;
    d_fn.execute(3, 5, &d_x, &d_y);
    std::cout << "Derivative of fn wrt d_x: " << d_x << "\n";
    std::cout << "Derivative of fn wrt d_y: " << d_y << "\n";
  }
```

Questions ?

Also wrote a blogpost to explain basics of automatic differentiation: https://vaithak.github.io/autodiff-clad/