CS207 Project: Milestone 1

@Software-Samurais

Kailas Amin
kailasamin@college.harvard.edu

Jingyuan Liu
jingyuanliu@g.harvard.edu

Erick Ruiz
eruiz@g.harvard.edu

Simon (Xin) Dong
xindong@g.harvard.edu

October 29, 2019

1 Introduction

The increasing importance of computational models in science and business alongside the slowing pace of advances in computing hardware has increased the need for efficient and accurate evaluations of derivatives. Many important applications such as simulation, optimization, and neural networks rely on repeated differentiation of complex functions.

Before the advent of automatic differentiation (AD) the primary method for derivative evaluation was the method of finite differences (FD), where the function to be evaluated is effectively treated as black box oracle.[1] As the FD method is effectively sampling, the granularity (i.e. step size) of the algorithm can introduce error effects if it is either too large or too small, but even at the perfect medium, $f'(x)$ evaluations cannot reach machine precision. The alternative approach, fully symbolic differentiation (SD), is cumbersome and inefficient in many cases. In the case of a complex computer program, the size of the symbolic expression could grow to outrageous size and cause significant inefficiency.

The approach of algorithmic differentiation seeks to find the best of both worlds, with machine precision and easy evaluation. This is done by repeated evaluation of the chain rule at a point stored in a table called the computational trace. Thus rather than storing to full symbolic expression, an AD code only needs to apply the chain rule to a specific evaluation, representable by a single variable. This approach allows us to achieve the accuracy of symbolic approaches while drastically reducing the cost of evaluation.

Within the umbrella of automatic differentiation, we seek to implement the forward mode which evaluates the intermediate results directly in an inside out manner. Other approaches such as reverse mode also have specific advantages especially in the fields of machine learning

[1] See the *Background* section for a more detailed introduction

and artificial intelligence– or in any context in which the number of inputs dominates the number of outputs.

The method of automatic differentiation, sometimes also referred to as algorithmic differentiation, addresses the weaknesses of the finite difference method by providing a systematic way to calculate derivatives numerically to arbitrary precision. The goal of `AutoDiff` is to implement the forward mode of automatic differentiation, as it is a relevant feature that even some mainstream machine learning libraries, such as PyTorch, lack.

2 Background

Understanding the concept of a derivative is crucial to all aspiring and practicing scientists, engineers, and mathematicians. It is one of the first concepts introduced in first-year calculus courses at all universities. The idea is simple. Given a function, $f(x)$, how can we quantify the rate of change of the function due to an infinitesimal change, Δx, in the argument, x? The answer is typically given in terms of the limit definition of the derivative.

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{1}$$

While equation (1) holds for any function, in practice, it is easier to calculate derivatives analytically according to a set of rules. However, obtaining an analytical expression for the derivative becomes exceedingly difficult if the function of interest is composed of many elementary functions. For example, consider the following function.

$$f(x) = \exp\left[\frac{\sqrt{x^3 - \ln x + \sin(4x^2)}}{\cos(3x^5)}\right] \tag{2}$$

Calculating the first derivative would result in the following expression.

$$\begin{aligned} f'(x) = \exp\left[\frac{\sqrt{x^3 - \ln x + \sin(4x^2)}}{\cos(3x^5)}\right] \sec^2(3x^5) \ldots \\ \times \left\{ \frac{\cos(3x^5)}{2\sqrt{x^3 - \ln x + \sin(4x^2)}} \left[3x^2 - \frac{1}{x} + 8x\cos(4x^2)\right] \ldots \right. \\ \left. + 15x^4 \sin(3x^5)\sqrt{x^3 - \ln x + \sin(4x^2)} \right\} \end{aligned} \tag{3}$$

Although feasible, successive calculations become more and more complex, and in practice, the quantity to be differentiated may not be a function in closed-form but rather a set of measurements or values given as a one-dimensional vector of numbers. In that case, equation (1) can be approximated using the finite difference method, which replaces an infinitesimal

change in the argument for a finite change. To show how this works, let us write the Taylor series expansion of an arbitrary function, $f(x)$, at the point $x + h$.

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \dots \tag{4}$$

Keeping only terms of $\mathcal{O}(h)$ leaves us with

$$f(x) \approx f(h) + hf'(x), \tag{5}$$

which we can rearrange to write an approximate expression for the derivative, $f'(x)$.

$$f'(x) \approx \frac{f(x + h) - f(x)}{h} \tag{6}$$

The finite change, h, is called the step size, and equation (6) is known as the forward difference. Its geometric interpretation is described in Figure 1.

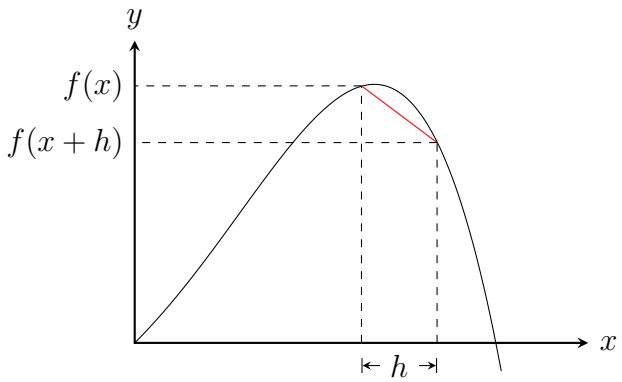

Figure 1: Geometric interpretation of the finite difference method. As the step size, h, decreases, so does the difference between $f(x + h)$ and $f(x)$. In principle, this should lead to a more accurate approximation to the true derivative, $f'(x)$. However, this is not always the case.

Although the finite difference method is useful and easy to implement, its accuracy can vary depending on the step size that is chosen. Suppose we wish to approximate the derivative of $f(x) = \ln x$ using the forward difference method described in equation (6) using step sizes $h = \{10^{-1}, 10^{-7}, 10^{-15}\}$. This is rather unnecessary because the analytical derivative is just $f'(x) = 1/x$, but this example will serve to illustrate the drawbacks of the finite difference method. At $h = 10^{-1}$, the numerical derivative is inaccurate because the step size is too large, making the calculations susceptible to truncation error. Conversely, at $h = 10^{-15}$, the forward difference method also gives inaccurate results because the calculations can only be represented to a finite precision by the hardware in use. Hence, rounding error also affects the stability of the finite difference method. Figure 2 summarizes the results.

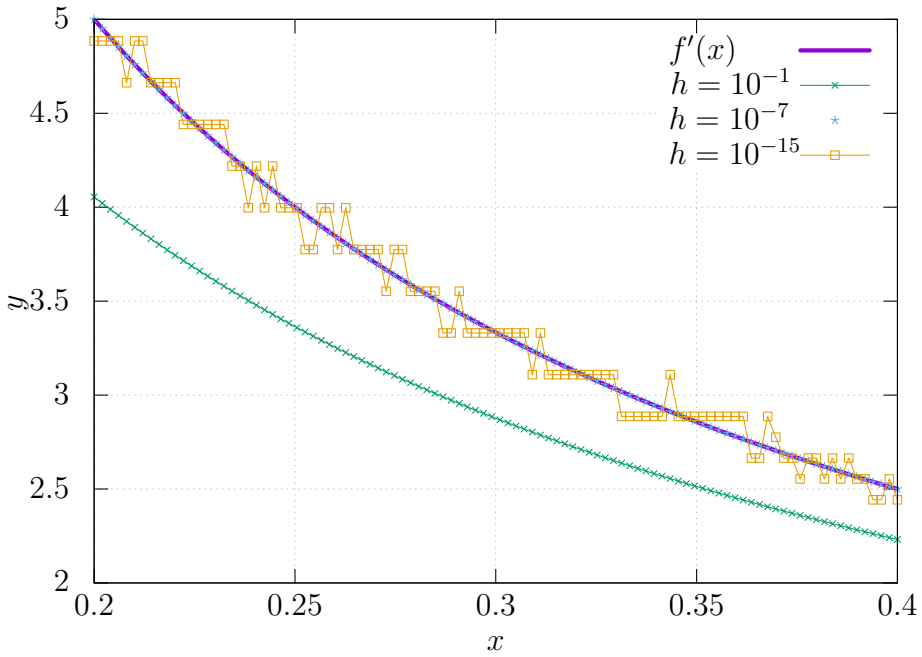

Figure 2: Accuracy of the finite difference method. The accuracy and stability of the approximate derivative actually gets worse with decreasing step size. The optimal step size for this case is $h = 10^{-7}$.

In order to evaluate the derivative via forward mode AD we first construct a computational graph which encodes the composition and dependence of sub-function evaluations. It is important to note that each elementary function evaluation needs to be understood at a symbolic level! After constructing the graph, the chain rule is simply applied successively to evaluations at a single point which then generates a table known as the computational trace. To illustrate the concept, consider the following example, adapted from *Evaluating derivatives: principles and techniques of algorithmic differentiation* by Griewank and Walther.

$$f(x, y) = \left[\sin\left(\frac{x}{y}\right) + \frac{x}{y} - \exp(y)\right]\left[\frac{x}{y} - \exp(y)\right] \tag{7}$$

A function of two arguments (i.e. variables x and y) like the one in equation (7) can be evaluated at point by replacing the arguments with numerical values. The series of calculations needed to carry out the evaluation can be visualized as a computation graph, as shown in Figure 3. The graph helps visualize the order of the computations, and it also establishes the dependence of successive calculations on previous ones. The procedure for constructing such a graph is the following. Define a node for each of the inputs using a new variable. For this example, we will let $x_1 = x$ and $x_2 = y$. From there, any successive nodes may accept two inputs at maximum, and each node represents a new calculation. For example, in the computational graph shown in Figure 3, the node x_3 represents the calculation x_1/x_2, which later becomes an input in successive nodes. While the computational graph is useful for visualizing the entire computation procedure, the computational trace is useful for storing

values as the computation is carried out. The computational trace for $f(1.5000, 0.5000)$ is given in Table 1.

Beyond the basic forward mode, there exists other implementations of AD in the so called "reverse mode". This approach can increase efficiency significantly in cases when the number of inputs is far greater than the number of outputs.

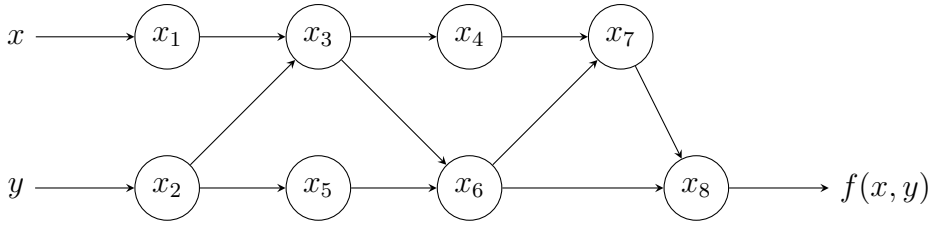

Figure 3: A computational graph. The function given equation (7) takes two inputs, x and y. Standard elementary operations are performed on the inputs to produce a single output, $f(x, y)$.

x_1	=	x	=	1.5000		
x_2	=	y	=	0.5000		
x_3	=	x_1/x_2	=	1.5000/0.5000	=	3.000
x_4	=	$\sin(x_3)$	=	$\sin(3.0000)$	=	0.1411
x_5	=	$\exp(x_2)$	=	$\exp(0.5000)$	=	1.6487
x_6	=	$x_3 - x_5$	=	3.0000-1.6487	=	1.3515
x_7	=	$x_4 + x_6$	=	0.1411+1.3513	=	1.4924
x_8	=	$x_7 \times x_6$	=	1.4924×1.3513	=	2.0167
f	=	x_8	=	2.0167		

Table 1: A typical computational trace using the example given in equation (7).

3 Usage Guide

3.1 Installation

Assuming that the user already has the latest version of Python and a package manager of choice installed, the first step towards using `AutoDiff` will be the installation. This can be accomplished using `conda` or `pip`, the two most popular package managers. This process will vary slightly, depending on the operating system, but essentially, the user will execute commands from the terminal to update the package manager and install `AutoDiff`.

```
# Using conda
conda update conda
conda install -c conda-forge AutoDiff
```

```
# Using pip
pip install AutoDiff
```

3.2 Getting Started

Once `AutoDiff` is installed, the user must import it to be able to use it. The user will have the option to either import the entire library or to choose only a subset of modules, classes, or methods to import. For instance, if the user only wishes to import the automatic differentiation class (and all of its methods) for linear functions of the form $f(x) = \alpha x + \beta$, then they will have the freedom to do so. It is widely accepted as good practice to use an alias when importing libraries. In the following example, the alias for `AutoDiff.linear` is just `linear`. Users will have the ability to define their own alias.

```
# Imports the necessary constructors and elementary functions
# (sin, exp, sqrt, etc.)
from AutoDiff import *

# Only imports forward mode automatic differentiation for linear functions
import AutoDiff.linear as linear
```

3.3 Example Usage

Here we use few simple examples to illustrate the API and data structures of the `AutoDiff` module.

```
import AutoDiff as ad

# A differentiable object instantiation
x = ad.Variable(3.0)
print(x.val, x.der)
# output: 3.0 0.0

y = x**2
print(y)
# output: AutoDiff(2.25, [3.0])

# What is dy/dx?
print y.der
# output: 3.0

# Specify both the function and derivative values
f = ad.linear(3.0, [2.1])

# Specify the function value only
```

```
# The derivative value will default to 1.0 for linear functions.
g = ad.linear(2.0)
```

Instances of other classes pertaining to `AutoDiff` would be defined in a similar manner. Once the user defines an instance of a class, they may perform standard elementary operations on that object. For example, `f*g` would return a new object whose function value is the product of the function values of `f` and `g` and whose derivative value is the product of the corresponding derivative values of `f` and `g`, according to the product rule.

$$\frac{d}{dx}(fg)\Big|_{x=a} = f'(x)g(x) + f(x)g'(x)\Big|_{x=a} \tag{8}$$

This way, the user would have the ability to calculate derivatives of composite functions using elementary operations while retaining arbitrary precision at each evaluation step. Suppose our expression is $z = x_1 \sin(x_2)$.

```
import numpy as np
# Multivariate Examples
x1 = ad.Variable(1.0)
x2 = ad.Variable(np.pi/2)
z = x1 * sin(x2)
print(z)
# output: AutoDiff(1.0, array[1.0, 1.0])
# array: dz/dx1, dz/dx2
```

As can be seen in the examples, when an `AutoDiff` object is printed out, we see two sets of numbers. The first is the current value. The next set of values are the derivatives of each variable, evaluated at the nominal value.

4 Software Organization

4.1 Directory Structure

/Ccs207-FinalProject

- /src — Back-end source code
  - /config — Configuration for the project
  - Auto_diff.py
  - forward.py
  - reverse.py
  - basic_funcs.py (add by Xin)
  - ...
- /gui — Front-end source code
  - /dist — Static css, js etc.
  - /template — Web html files

/img	Images used for font-end
/utils	Preprocessing scripts
input_parser.py	
...	
/test	Test cases
test_forward.py	
test_reverse.py	
test_eval.py	
...	
/doc	Documentation and records
milestone1.tex	
milestone2.tex	
...	
__init__.py	Initialization
requirements.txt	Packages on which the program depends
README.md	Introduction for the project

4.2 Modules and functionality

We plan to include:

- AutoDiff module for definition of the AutoDiff class.
- Forward module for forward mode in automatic differentiation.
- Reverse module for reverse mode in automatic differentiation.
- Some Utils modules for parsing the input, preprocessing and start main program.

4.3 Test Suite

Coding is the fundamental part of software development. Equally significant is build and testing. We would utilize *Travis CI* and *CodeCov* to make the development process more reliable and professional. The test suite will be placed in the test folder.

- *Travis CI* is used as a distributed CI (Continuous Integration) tools to build and automate test the project.
- *CodeCov* is used for test results analysis (eg. measuring test code coverage) and visualization.

4.4 Software Package and Distribution

- Package distribution
  We will package our software using PyPI (Python Package Index) for release. Write

and run 'setup.py' to package the software and upload it to the distribution server, thus people in community could easily download our package by 'pip install'.

- Version Control
  We will take Version Control into consideration according to the standard in Python Enhancement Proposal (PEP) 386. With version control, we can tell the user what changes we made and set clear boundaries for where those changes occurred.

- Framework
  - For web development, we would use *Flask*, a micro web framework, which is suitable for a small team to complete the implementation of a feature-rich small website and easily add customized functions.
  - For GUI (Graphical User Interface), we may choose Vue.js, a JavaScript framework for building user interfaces and single-page applications. Because it offers many API (Application Program Interface) to integrate with existing projects and is easy to get started. It is better in code reuse compared to frameworks like jQuery.

5 Implementation

Reverse Mode and *Forward Mode* will be implemented separately with different data structures but shared basic functions like `Add`, `Mul` in the file *basic_funcs.py*.

5.1 Reverse Mode

Data Structure: The first step is to parse the input equation into a computational graph for the forward pass. Every leaf variables (aside from constants) is an instance of `Node`. Computation between leaf variables will be reloaded and return a new node. By doing this, we can parse the input equation into a computational graph automatically.

Classes and attributes: Every node in the computation graph is an instance of class `Node` which includes `self.inputs`, the input nodes of current node.
`compute(self, node, input_vals)` and `gradient(self, node, output_grad)`.

Methods: Since the order of gradient computing is crucial for *Reverse Mode*, we will use a simple algorithm which does a post-order Depth-First-Search (DFS) traversal on the given nodes in the computational graph, going backwards based on input edges. With post-order DFS, a node will be added to the ordering after all its predecessors are traversed and we can get a topological sort finally. Then, we can reverse the topological sort list of nodes and start from the output nodes to compute gradient. Pseudo codes are shown as follows,

```
def gradient(out):
    node_to_grad = {}
    nodes = get_reverse_topo_order(out)
    for node in nodes:
        # get gradient of current node
        grad = sum(partial_adjoints from output_edges)
        # get partial gradient of input nodes of current node
        input_grads = [node.gradient(i, grad) for i in node.inputs]
        add input_grads to node_to_grad
    return node_to_grad
```

External dependencies: Numpy is used to support operations between matrix like `MatMul`.

Elementary Functions: We will implement them inside the class of `Node`. For example, for the method of `__add__`, we can implement it as follows,

```
class Node(object):
    def __add__(self, other):
        if isinstance(other, Node):
            new_node = add_op(self, other)
        else:
            new_node = add_byconst_op(self, other)
        return new_node
     # Allow left-hand-side add and multiply.
    __radd__ = __add__
```

5.2 Forward Mode

Forward Mode is easier than *Reverse Mode* to implement with different data structure.

Data Structure: We will use `STACK` as the main data structure. Basically, each equation can be computed sequentially with Reverse Polish notation (RPN). For each node, we have to carry derivatives along with evaluation flow. In addition, we have to save the whole graph because we do not know whether we will use them again.

Classes and attributes: We will use the class of `Node` again. However, we don't have to maintain the whole graph this time. Following the order of RPN, a new node will be added with two main attributes, `value` and `derivation`.

It is worth mentioning that `derivation` of each node is a list with the same length. Specifically, the first element of the list is the derivation of the first leaf variable with respect to the current node. In addition, `derivation` is computed based on the type of gradient and the input gradient.

6 Bibliography

Andreas Griewank. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. 2000.