Nov. 19, 2020

Group 16

Team members: Connor Capitolo, Haoxin Li, Kexin Huang, Chen Zhang

CS 107 Final Project Milestone 2

1. Introduction

Differentiation is one of the core applications in many different scientific fields. At a basic level, the derivative of a function $y = f(x)$ of a variable x is a measure of the rate at which the value y of the function changes concerning the change of the variable x.[1] Whether it is using gradient descent or backpropagation in machine learning to modeling populations or even epidemics like COVID-19 in biology, differentiation is a key concept that is used. Derivatives are so important that there are college-level courses dedicated to its understanding and application. With the knowledge that derivatives are important in many different fields (and continuing to become even more important with the increase in computational power and resources), the question then becomes the best way to calculate derivatives.

There are currently three main techniques to compute the derivative: finite difference, symbolic differentiation, and automatic differentiation. While the finite difference method is quick and easy to implement (making it still a very useful tool), it suffers from machine precision and rounding errors. One of our goals is to calculate the derivative at machine precision; enter symbolic differentiation, which is able to solve this problem. However, symbolic differentiation becomes too computationally expensive when the function(s) start to get very complex. Automatic differentiation solves the problem of both computational cost and machine precision, making it the perfect technique for finding the derivative.[2]

---

[1] "Derivative." *Wikipedia*, Wikimedia Foundation, 15 Oct. 2020, en.wikipedia.org/wiki/Derivative.

[2] Brown, Lindsey, and David Sondak. "An Introduction to Automatic Differentiation with a Visualization Tool." *Auto-ED*, 2019, auto-ed.readthedocs.io/en/latest/index.html.

2. Background

At the core of automatic differentiation is the chain rule. Taking the function f(t) = h(u(t)), here it is in its most basic form:

$$\frac{\partial f}{\partial t} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial t}$$

A quick example is $\cos(9t^2)$. Performing the chain rule, we obtain $-18t*\sin(9t^2)$.

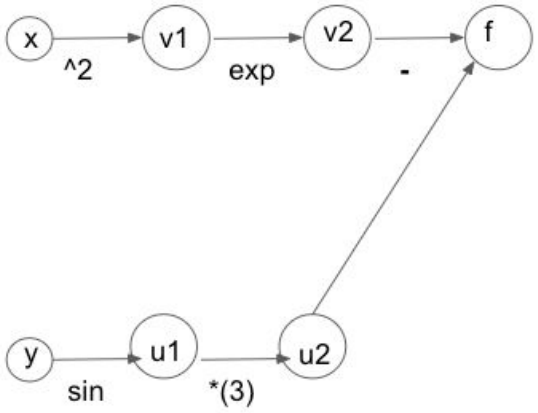
Any complex function can be broken down into its elementary operations and elementary functions. Elementary operations include addition, subtraction, multiplication, and division, while elementary functions include exponents, logs, trigonometric functions (sin, cos, tan), and hyperbolic trigonometric functions (sinh, cosh, tanh). As we will see in the example shortly, we can build more complex functions from these elementary functions and operations, while using the chain rule to obtain the derivative.

There are two forms of automatic differentiation: forward mode and reverse mode. Forward mode works best when there is a large number of objective functions to evaluate and only a few independent variables (n >> m), while reverse mode is best when the number of inputs is much larger than the number of functions (m >> n).

Since the basic requirement of this library is to implement forward mode, let's examine a somewhat basic example that can be easily extended to include more variables and more functions.

$$f(x, y) = \exp x^2 - 3*\sin(y)$$

This function consists of two independent variables (x,y) and one function. The easiest way to start and understand how to obtain the derivative for this complex function is to visualize a computational graph, as shown below:



Looking at the above graph from left to right, you can see that elementary operations are being performed on the variables x and y until eventually we can reach a final output by working inside-to-out.

Let's slowly step through the computational graph to understand exactly how it works. Starting with the x variable, we first square it and then take the exponent. Looking back to the function f(x,y), these two elementary operations compose $\exp(x^2)$, which can be thought of as a new variable, v2. Now looking at the y variable, we first take the sin and then multiply by three. Looking back to the function f(x,y), these two elementary operations compose 3*sin(y), which can be thought of as a new variable, u2. The final step is to take the two variables composed of elementary operations, v2 and u2, and subtract to find the output value.

While this is a nice visual that shows how to obtain the function value, it doesn't really show how to obtain the function gradient at a specific point. This is where the evaluation trace comes into play.

Here is the evaluation trace for the point (2,4):

Trace	Elementary Operation	Numerical Value	Elementary Derivative	∇_x	∇_y
x	x	2	$\dot{x}$	1	0
v_1	x^2	4	$2x\dot{x}$	4	0
v_2	$\exp(v_1)$	54.598150033144236	$\exp(v_1)\dot{v_1}$	218.39260013257694	0
y	y	4	$\dot{y}$	0	1
u_1	$\sin(y)$	-0.7568024953079282	$\cos(y)\dot{y}$	0	-0.6536436208636119
u_2	$3u_1$	-2.2704074859237844	$3\dot{u_1}$	0	-1.960930862590836
f	$v_2 - u_2$	56.86855751906802	$\dot{v_2} - \dot{u_2}$	218.39260013257694	1.960930862590836

Based on the results for this, we can see that the function value is ~56.869, the gradient of the function f with respect to x is ~218.393, and the gradient of the function f with respect to y is ~1.961.

One important point to note here is that the seed vector was chosen to be p = (1,0) for the gradient of f with respect to x, and p = (0,1) for the gradient of f with respect to y. Carefully choosing the seed vectors in this way guarantees that we get the actual value of the derivative output.

2.1. Gradient and Jacobian

If we have $x \in R^m$ and function $h(u(x), v(x))$, we want to calculate the gradient of h with respect to x:

$$\nabla x\, h = \frac{\partial h}{\partial u} \nabla x\, u + \frac{\partial h}{\partial u} \nabla \mathrm{x}\, v$$

In the case where we have a function $h(x) : R^m \longrightarrow R^n$, we write the Jacobian matrix as follows, to store the gradient of each output with respect to each input.

$$\mathbf{J} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_m} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial x_1} & \frac{\partial h_n}{\partial x_2} & \cdots & \frac{\partial h_n}{\partial x_m} \end{bmatrix}$$

In general, if we have a function g (y (x)) where $y \in R^n \text{ and } x \in R^m$. Then g is a function of possibly n other functions, each of which can be a function of m variables. The gradient of g is now given by

$$\Delta x\, g = \sum_{i=1}^{n} \frac{\partial g}{\partial y_i} \nabla_x\, y_i\,(x)$$

2.2. Elementary Functions

An elementary function is built up of a combination of constant functions, operations, algebraic, exponential, trigonometric, hyperbolic and logarithmic functions and their inverses under repeated compositions.

Elementary Functions	Example
powers	x^2
roots	$\sqrt{x}$
exponentials	e^x
logarithms	$\log(x)$
trigonometrics	$\sin(x)$
inverse trigonometrics	$\arcsin(x)$
hyperbolics	$\sinh(x)$

3. How to Use *apollo-ad*

3.1. Package Installation, testing and import

- Before the installation, setup a virtual environment for apollo_ad:

```
# Install virtualenv
pip install virtualenv
pip install pytest
# Create virtual environment
virtualenv apollo_ad_env
# Activate virtual environment
source apollo_ad_env/bin/activate
```

We provide three ways for users to install the package:

A. Download from GitHub

- You can directly download from Github to your folder via these commands in the terminal:

```
git clone https://github.com/West-Coast-Quaranteam/cs107-FinalProject.git
cd cs107-FinalProject
pip install -r requirements.txt
```

- Test the package

Run module test in apollo_ad/tests/

```
pytest test_scalar.py
```

3.2. import functions

To use the apollo_ad, import the Variable class in the apollo_ad package, but first we have to import the path of the package

```
import sys
sys.path.append('../apollo_ad/') #relative path to the apollo_ad
from apollo_ad import *
```

3.3. Scalar, single function

We currently only support scalar and single functions for all the elementary functions. We show here by an example. First, we want to calculate the derivative of x for function y = cos(x) + x^2. We need to first define x and its initial value:

```
x = Variable(2) # instantiate a variable x, class variable
```

Note that, in default, the seed of the derivative is set to 1. Then, you can specify the function:

```
y = Variable.cos(x) + x ** 2 # class fct, overwrite the dunder method
```

Notice that we define cos function as a static function in the Variable class. Thus, to call cos, use Variable.cos(.). This is also the case for other trig and exponential functions. For squares, we use the dunder method __pow__. Then, we calculate the value y based on these operations. The output y is also a Variable object with updated var and der, which are calculated accordingly. To check whether or not this is the case:

```
# y.var: 3.583853163452857 y.der: [3.09070257]
assert y.var == np.cos(2) + 4
assert y.der == -np.sin(2) + 2 * 2
```

We see that it indeed calculates the correct value and derivative.

For the second example, in addition to applying elementary functions on one single variable, we can also do constants. We showcase it through another function y = 2 * log(x) - sqrt(x) / 3. To calculate the derivative of y on x, we can do similar operations as above:

```
x = Variable(2)
y = 2 * Variable.log(x) - Variable.sqrt(x)/3
# Value: 0.9148898403288588 , Der: [0.88214887]
assert np.around(y.var, 4) == np.around(2 * np.log(2) - np.sqrt(2)/3,
4)
assert np.around(y.der, 4) == np.around(2 * 1/2 - 1/3 * 1/2 *
2**(-1/2), 4)
```

4. Software Organization

4.1. Directory structure

```
cs107-FinalProject/
            README.md
            LICENSE
            .travis.yml
            requirements.txt
            setup.py
            setup.cfg
            apollo_ad/
                __init__.py
                Apollo_ad.py
                tests/
                     __init__.py
                     test_scalar.py
            docs/
                Milestone1.pdf
                Milestone2_progress.pdf
                Milestone2.pdf
```

4.2. Basic modules and functionality

apollo_ad/apollo_ad.py: Everything at the moment is encapsulated in this python file. The main class is `Variable` where it can keep track of the derivatives of each variable in the computation. All the operations, including the basic ones (addition, division, multiplication etc.) and more advanced ones (exponential, trigonometry). We created basic operation methods as instance methods as we are overloading the dunder methods. Advanced methods are created as static methods, and eventually we plan to implement them into a seperate class dedicated for them.

4.3. Test suite

Our test suite is in a test file called *test.py* and is in *apollo_ad*/*tests* folder.

We use TravisCI for automatic testing for every push, and Coveralls for line coverage metrics.

These integrations have already been set up, with badges included in the *README.md*. Users could run the test by navigating to the *tests/* folder and running the command *pytest test.py* from the command line.

4.4. Implementation

__init__.py

```
Empty for now.
```

apollo_ad.py

```
# Instances of the variable class represent independent variables. We re-defined the dunder methods for elementary operations. It can take in a Python int or float object.
class Variable:
    Attributes:
        val: value of the obj unit
    Methods:
        __init__(self, value)
        __eq__(self, other)
        __ne__(self, other)
        __neg__(self, other)
        __add__(self, other)
        __sub__(self, other)
        __radd__(self, other)
        __rsub__(self, other)
        __mul__(self, other)
        __rmul__(self, other)
        __div__(self, other)
        __rdiv__(self, other)
        __rtruediv__(self, other)
        __rfloordiv__(self, other)
        __gt__(self, other)
        __le__(self, other)
```

```
        __lt__(self, other)
        __ge__(self, other)
        __abs__(self)
        __pow__(self, other)
        __rpow__(self, other)
        __repr__(self, other)
        __str__(self, other)
        @staticmethod
        sqrt(variable)
        exp(variable)
        log(variable)
        sin(variable)
        cos(variable)
        tan(variable)
        arcsin(variable)
        arccos(variable)
        arctan(variable)
        sinh(variable)
        cosh(variable)
        tanh(variable)
```

As for now, there are only two attributes for the `Variable` type. The class takes two user inputs, `var` and `der` and are assigned to `self.var` and `self.der`. `self.var` stores the numerical value of the variable. `self.der` stores the the derivative, and the seed is defaulted to 1, so it will be initialized to 1 unless the user specifies otherwise. Under this class, we overloaded all the dunder methods that are responsible for basic operations such as addition, subtraction, multiplication, etc. Also under this Variable class are more advanced operations such as exponential and trigonometric functions. These methods are static as they are not instance dependent and can be used as what Numpy functions would without keeping track of the derivatives. At the current

stage, we expect the user to pass in integers or floats to the constructor and all the math functions. The user can create a `Variable` instance by passing in a string, but it would not be able to perform any math operations. In the later implementation, we will forbid the user to instantiate a `Variable` object using anything other than an integer or float in the constructor.

We have implemented all the math functions to perform any basic operations, but they only support scalar variables and functions. In the next step, we need to expand our implementation so that they can handle multi-variable functions and the user can accurately calculate the Jacobian of each variable respective each function as desired. More specifically, we want to create another Class type called `Array` that inherits from `Variable` class that can handle vector inputs and functions.

5. Future Features

5.1. Reverse Mode

The main additional feature that we are looking to implement is reverse mode. While the forward and reverse mode will both allow for calculating the derivative and will provide the same answer, depending on the type of problem at hand, one technique may be drastically faster than the other. A general rule is when the number of inputs is relatively small compared to the number of outputs (n<<m), the forward mode is the preferred technique since there will be fewer operations performed, making it more computationally efficient. On the other hand, when the number of inputs is large compared to the number of outputs (n>>m), the reverse mode is the preferred technique since there will be fewer operations performed, making it more computationally efficient[3]. A special case of reverse mode is the backpropagation algorithm, where there is a single scalar output typically seen as the loss function. Backpropagation is the foundational component for why neural networks are so popular today since they can efficiently model very nonlinear functions.

Since reverse mode requires performing the partial derivative calculations and chain rule on the backward pass, one of the first challenges that we face is an efficient way to create and store the

[3] https://math.stackexchange.com/questions/2195377/reverse-mode-differentiation-vs-forward-mode-differentiation-where-are-the-be

graph (something we did not do for our forward mode implementation). The other major challenge that we've identified is how we will handle performing the chain rule on the partial derivatives when a node has more than one child. Our current belief is that if we can generate an approach that effectively stores the computational graph, handling of nodes with multiple children will fall into place.

Our plan right now is to create a separate package within *apollo_ad* that performs the reverse mode. While there is certainly some overlap between reverse mode and forward mode, especially for our initial implementation we believe it's best to treat the reverse mode separate from forward mode. While this may create more work than is necessary to implement, this will serve as a great learning tool for both our team as well as the users of *apollo_ad*. Since we've struggled with understanding the differences in computational efficiency between forward mode and reverse mode, one of the options we hope to provide is the ability for a user to see how long it takes to run reverse mode vs. forward mode. This educational tool will help users better intuitively understand when to use each of the different automatic differentiation implementations.

Another potential feature when implementing our reverse mode is creating a script that decides whether it is more efficient to use reverse mode or forward mode based on the user's input. We recognize that this may be outside the scope of the project due to time constraints as well as (having a good boundary to identify when to use forward mode or backward mode will almost certainly require additional machine learning techniques), but it wouldn't be too difficult to create a simple threshold that could then be communicated to the user if our program is using reverse mode or forward mode to provide the derivative output.