Nov. 2, 2020

Group 16

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

CS 107 Final Project Milestone 1

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 with respect to 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:

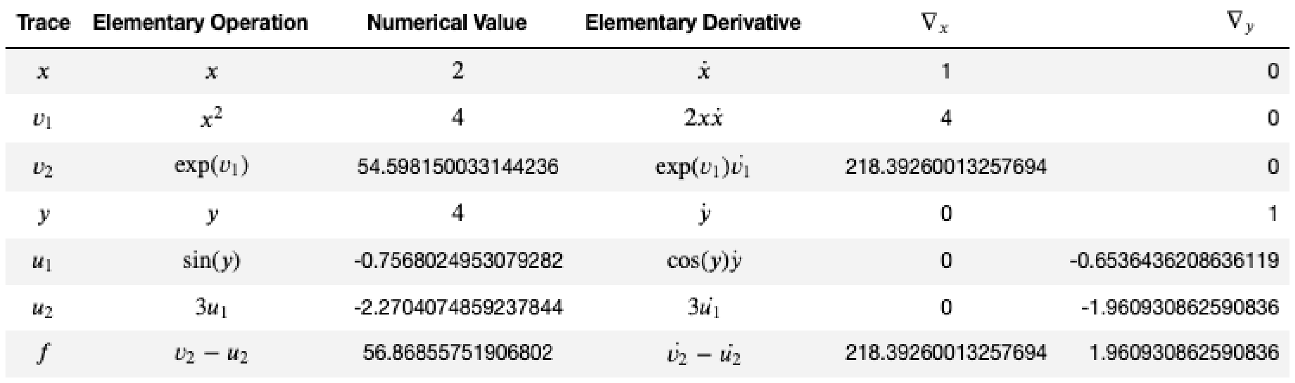
$$\begin{tikzcd}
x \arrow[r, "{}^\wedge 2"'] & v1 \arrow[r, "\exp"'] & v2 \arrow[r, "-"'] & f \\
y \arrow[r, "\sin"'] & u1 \arrow[r, "*(3)"'] & u2 \arrow[ur] &
\end{tikzcd}$$

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 $y \in R^n$ *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. PIP install and import functions

```
pip install apollo_ad
```

```
from apollo_ad import utils
from apollo_ad import variable
from apollo_ad import vector
from apollo_ad import fct
```

3.2. Scalar, single function

```
x = variable(value = 2) # instantiate a variable x, class variable
y = variable(value = 3) # instantiate a variable x, class variable
f = fct(value = [2 * x + 3 * y^2 + utils.exp(x) + utils.sin(y)]) #
class fct, overwrite the dunder method *, +, ^2

assert f.derivative(x) == 2 + np.exp(x.val)
assert f.derivative(y) == 6 * y + np.cos(y.val)
assert f.gradient() == np.array([2 + np.exp(x.val), 6 * y +
np.cos(y.val)])
```

3.3. Scalar, multiple functions

```
x = variable(value = 2) # instantiate a variable x, class variable
y = variable(value = 3) # instantiate a variable x, class variable
f = fct(value = [2 * x + 3 * y^2 + utils.exp(x) + utils.sin(y), 3 * x
+ 4 * y]) # class fct, overwrite the dunder method *, +, ^2

assert f.derivative(x) == np.array([2 + np.exp(x.val), 3])
assert f.derivative(y) == np.array([6 * y + np.cos(y.val), 4])
assert f.gradient() == np.array([[2 + np.exp(x.val), 3],
                                  [6 * y + np.cos(y.val), 4]])
```

3.4. Vector, single function

```
v1 = vector(value = [2, 3])) # instantiate a vector v1, class vector
v2 = vector(value = [1, 9])) # instantiate a vector v2, class vector

f = fct(value = [3 * v1 + utils.sin(v2)])
assert f.derivative(v1) == np.array([3, 3])
assert f.derivative(v2) == np.array([np.cos(1), np.cos(9)])
assert f.gradient() == np.array([[3, 3],
                                 [np.cos(1), np.cos(9)]])
```

3.5. Vector, multiple functions

```
v1 = vector(value = [2, 3])) # instantiate a vector v1, class vector
v2 = vector(value = [1, 9])) # instantiate a vector v2, class vector

f = fct(value = [3 * v1 + utils.sin(v2),  2 * v2 + utils.exp(v1)])

assert f.derivative(v1) == np.array([[3, 3], [np.exp(2), np.exp(3)]])
assert f.derivative(v2) == np.array([[np.cos(1), np.cos(9)], [2, 2]])
assert f.gradient() == np.array([[[3, 3], [np.exp(2), np.exp(3)]],
                                 [[np.cos(1), np.cos(9)], [2, 2]]])
```

4. Software Organization

4.1. Directory structure

```
cs107-FinalProject/
            README.md
            LICENSE
            .travis.yml
            requirements.txt
            setup.py
            setup.cfg
            apollo_ad/
                __init__.py
                utils.py
                apollo_ad.py
            docs/
                Milestone1.pdf
            tests/
                test.py
```

4.2. Basic modules and functionality

apollo_ad.utils.py: it contains helper functions of the package, such as derivative modules for nonstandard elementary functions (sqrt, log, …).

apollo_ad.apollo_ad.py: it contains the core data structure, classes, attributes, methods for the AD functionalities.

4.3. Test suite

Our test suite is in a test file called *test.py* and is in its own *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. Packaging and distribution

We provide two ways for users to install our packages.

Before the installation, setup a virtual environment for apollo_ad:

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

Now, 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
```

Alternatively, use pip install:

```
pip install apollo_ad
```

We also utilized the Python Package Index (PyPI) for distributing our package. PyPI is the official third-party software repository for Python.

4.5. Additional Capabilities

(Lowest priority) We will add a UI component if time allows; details to be determined.

5. Implementation

__init__.py

```
from .apollo_ad import *
from .utils import *
```

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)
         __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)
         __pow__(self, other)
         __rpow__(self, other)

# The vector class inherits from the variable class. It lets users
define vector variables by taking in a numpy array of variable
instances. We re-defined the dunder methods for vector/matrix
operations.
class vector(variable):
      Attributes:
          var : np.array
      Methods:
          __init__(self, value:np.array)
         __eq__(self, other)
         __neg__(self, other)
         __add__(self, other)
         __sub__(self, other)
         __radd__(self, other)
         __rsub__(self, other)
         __rmatmul__(self, other)
         __rmul__(self, other)
         __div__(self, other)
         __rdiv__(self, other)
         __gt__(self, other)
         __le__(self, other)
         __lt__(self, other)
         __ge__(self, other)
         __pow__(self, other)
         __rpow__(self, other)
         __rdivmod__(self, other)
         __rmod__(self, other)

# an instance of the fct class takes in an operation or a vector of
operations, creates the trace table and hosts the methods for
derivative/gradient calculations.
class fct:
      Attributes:
          var : variable
          trace: list
      Methods:
          __init__(self, value: variable/vector):
          derivative(self, variable/vector):
          gradient(self):

# Helper for dunder methods for elementary operations such as +,-,*,/
etc...
class addition:
    Attributes:
        var1
        var2
    methods:
        __init__(self, var1, var2)
        compute(self)
        derivative(self, var)

class Subtraction:
    Attributes:
        var1
        var2
    methods:
        __init__(self, var1, var2)
        compute(self)
        derivative(self, var)


class Multiplication:
    Attributes:
        var1
        var2
    methods:
        __init__(self, var1, var2)
        compute(self)
        derivative(self, var)

class Division:
    Attributes:
        var1
        var2
    methods:
        __init__(self, var1, var2)
        compute(self)
        derivative(self, var)

class Power:
    Attributes:
        var1
        var2
    methods:
        __init__(self, var1, var2)
        compute(self)
        derivative(self, var)
```

utils.py

```
# This file holds all the classes for special operations like sin,
cos, exp, sqrt etc.
class sin:
    Attributes:
        var1
    methods:
        __init__(self, var1)
        compute(self)
        derivative(self, var1)

class cos:
    Attributes:
        var1
    methods:
        __init__(self, var1)
        compute(self)
        derivative(self, var1)
class log:
    Attributes:
        var1
    methods:
    __init__(self, var1)
    compute(self)
        derivative(self, var1)
class exp:
    Attributes:
        var1
    methods:
        __init__(self, var1)
        compute(self)
        derivative(self, var1)
class sqrt:
    Attributes:
        var1
    methods:
        __init__(self, var1)
        compute(self)
        derivative(self, var1)
```

6. Feedback

6.1. Milestone 1

>> *the UI component should be the least priority*

Our focus will be on implementing forward mode and reverse mode. If time permits, we will be adding the UI component as well.