Dec. 12, 2020

Group 16

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

Apollo_ad Documentation

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 efficient 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 (still making it a 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.

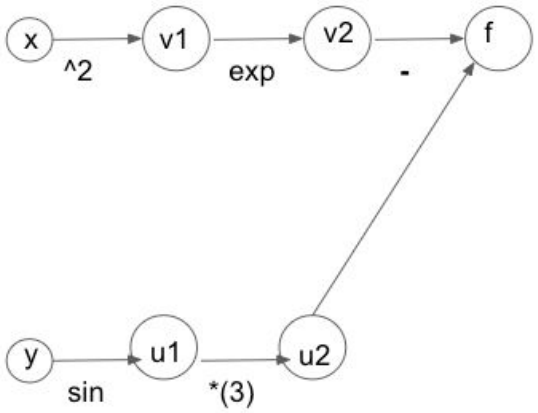
There are two forms of automatic differentiation: forward mode and reverse mode. Forward mode works best when there is a large number of objective functions to evaluate and only a few independent variables (n >> m), while reverse mode is best when the number of inputs is much larger than the number of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

2.1. Forward Mode

Since the basic requirement of this library is to implement forward mode, let's examine a somewhat simple 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 above function at 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.

2.2. Reverse Mode

While the reverse mode will obtain the exact same derivatives at the given point as the forward mode, it's implementation is slightly different. On the forward pass, rather than performing the chain rule as is done with forward mode, reverse mode will store the partial derivatives. We can look at two of the most important and widely used elementary operations to understand this: multiplication and addition. For multiplication, if $v_1 = x_1 x_2$ is a node, then we will store $\frac{\partial v_1}{\partial x_1}$ (in this case equal to x2) and $\frac{\partial v_1}{\partial x_2}$ (in this case equal to x1). For addition, if $v_2 = x_3 + x_4$, we would store $\frac{\partial v_2}{\partial x_3}$ (in this case equal to 1) and $\frac{\partial v_2}{\partial x_4}$ (in this case equal to 1). Similar to how we have a seed vector for the forward mode, we have one as well for reverse mode. Taking v_N as the output variable, we set $\frac{\partial f}{\partial v_N}$ equal to 1; while we can set the seed to whatever value we'd like,

setting it to 1 allows us to correctly calculate the derivatives. Having stored all these values and knowing the computational graph, we can then calculate the derivatives with respect to the function and perform the chain rule on the backward pass.

Here's the main equation for reverse mode:

$$\frac{\partial f}{\partial u_i} = \sum_{\text{j a child of i}} \frac{\partial f}{\partial u_j} \frac{\partial u_j}{\partial u_i}$$

At each node/variable (whether an intermediate or an input), we can calculate its derivative with respect to the function f. Therefore, we start with the nodes that point directly to the output, calculate their derivatives, and then continue to work backwards until we get to the inputs. The only difficulty is when we get to a branch in the graph, but that is why there is a summation symbol in the graph above since all we need to do is sum the different paths in order to get the derivative for that specific node.

2.3. 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

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

The forward mode calculates the Jacobian-vector product, $\boldsymbol{Jp}$, where p is the seed vector.

The reverse mode calculates the transpose-Jacobian-vector product, $\boldsymbol{J^T p}$, where p is the seed vector.

2.4. 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

Installation (we assume you are familiar with virtual environments and Git)

apollo-ad is available at https://pypi.org/project/apollo-ad/. To download, simply type the below command in the terminal:

```
pip install apollo-ad
```

To install from source and test it:

```
install virtualenv
pip install virtualenv
# create virtual environment
virtualenv apollo_ad_env
# activate virtual environment
source apollo_ad_env/bin/activate
# clone from GitHub
git clone https://github.com/West-Coast-Quaranteam/cs107-FinalProject.git
# get into the folder
cd cs107-FinalProject
# install requirements
pip install -r requirements.txt
# test
pytest apollo_ad/tests/
```

3.2. import functions

To use apollo_ad, import all the classes in the apollo_ad package

```
from apollo_ad import *
```

3.3. Programming Usage

apollo_ad expects two inputs: a dictionary with variable name as the key and variable value as the value, as well as a list of strings where each string describes a function:

```
from apollo_ad import auto_diff
var = {'x': 0.5, 'y': 4}
```

```
fct = ['cos(x) + y ** 2',
       '2 * log(y) - sqrt(x)/3',
       'sqrt(x)/3',
       '3 * sinh(x) - 4 * arcsin(x) + 5']

out = auto_diff(var, fct)
print(out)

# -- Values --
# Function F1: 16.87758256189037
# Function F2: 2.5368864618442655
# Function F3: 0.23570226039551587
# Function F4: 4.468890814088047
# -- Gradients --
# Function F1: [-0.47942554  8.        ]
# Function F2: [-0.23570226  0.5       ]
# Function F3: [0.23570226 0.        ]
# Function F4: [-1.23592426 -4.61880215]
```

apollo_ad supports both forward and reverse mode in the backend, where the auto_diff class automatically detects which is the best way to use, depending on the number of inputs and outputs. You can also directly use the Forward and Reverse mode class. Remember that both forward and reverse mode should produce the same results.

```
from apollo_ad import Forward, Reverse
var = {'x': 3, 'y': 4}
fct = ['cos(x) + y ** 2', '2 * log(y) - sqrt(x)/3']
out = Forward(var, fct)
print(out)

# -- Values --
# Function F1: 15.010007503399555
# Function F2: 3.2938507417182654
# -- Gradients --
# Function F1: [-0.14112001  8.        ]
# Function F2: [0.23710829 0.5       ]

var = {'x': 1, 'y': 2}
fct = ['x * y + exp(x * y)', 'x + 3 * y']
out = Reverse(var, fct)
print(out)
```

```
# -- Values --
# Function F1: 9.38905609893065
# Function F2: 7.0
# -- Gradients --
# Function F1: [16.7781122  8.3890561]
# Function F2: [1. 3.]
```

You can also specify the seed. As the seed in forward mode is for each variable whereas in reverse mode it's for each function, we expect a dictionary for forward mode and a list for reverse mode. Here is an example:

```
var = {'x': 3, 'y': 4}
seed = {'x': 1, 'y': 2}
fct = ['cos(x) + y ** 2', '2 * log(y) - sqrt(x)/3']
z = auto_diff(var, fct, seed)
# -- Values --
# Function F1: 15.010007503399555
# Function F2: 2.1952384530501554
# -- Gradients --
# Function F1: [-0.14112001 16.        ]
# Function F2: [-0.09622504  1.        ]
```

3.4. Interface Usage

We also provide a nice interface, where you can specify the variable values and functions without any coding:

```
from apollo_ad import UI
UI()
```

It would provide prompts for users to enter the variable values and functions and prints out the gradients and function values. A sample UI is provided here:

```
Welcome to Apollo AD Library!
Enter the number of variables:
2
Enter the number of functions:
3
Type the variable name of variable No. 1 (Please only input a name that CANNOT be
cast as an integer or float):
```

```
a
Type the value of variable a (Please only input a float):
3
Type the derivative seed of variable a (Please only input a float; your default
input should be 1):
1
Type the variable name of variable No. 2 (Please only input a name that CANNOT be
cast as an integer or float):
b
Type the value of variable b (Please only input a float):
2
Type the derivative seed of variable b (Please only input a float; your default
input should be 1):
1
Type function No. 1 :
a + b + sin(b)
Type function No. 2 :
sqrt(a) + log(b)
Type function No. 3 :
exp(a * b) + a ** 2
---- Summary ----
Variable(s):
{'a': '3', 'b': '2'}
Function(s):
a + b + sin(b)
sqrt(a) + log(b)
exp(a * b) + a ** 2
---- Computing Gradients ----
# of variables < # of functions ====> automatically use the forward mode!
---- Output ----
-- Values --
Function F1: 5.909297426825682
Function F2: 2.4251979881288226
Function F3: 412.4287934927351
-- Gradients --
Function F1: [1.         0.58385316]
Function F2: [0.28867513 0.5       ]
Function F3: [ 812.85758699 1210.28638048]
```

4. Software Organization

4.1. Directory structure

```
cs107-FinalProject/
```

```
README.md
MANIFEST.in
LICENSE
.travis.yml
.codecov.yml
requirements.txt
setup.py
setup.cfg
apollo_ad/
    __init__.py
    apollo_ad.py
    UI.py
    demo.py
    tests/
         __init__.py
         test_forward_mode.py
         test_reverse_mode.py
docs/
    README.md
    Milestone1.pdf
    Milestone2_progress.pdf
    Milestone2.pdf
    Final_Milestone.pdf
```

4.2. Basic modules and functionality

apollo_ad/apollo_ad.py: Major functions are encapsulated in this python file. There are five main classes:

-- 'Variable': It is the variable class for forward mode and includes the derivative calculations for 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. It only works for one function.

-- 'Reverse_Mode': It is the variable class for reverse mode and includes derivative calculations for all the operations. It stores the computation graph and would conduct a recursive derivative computation when called for gradients. It only works for one function.

-- ‘Forward’: it is a wrapper of the class Variable where it extends to support multi-functions input. It takes in the input functions to be a list of strings and a dictionary of variables names and values. It then outputs the gradients and values for each function and variables. It automatically supports both multi-functions/variables.

-- ‘Reverse’: it is a wrapper of the class Reverse_mode where it extends to support multi-functions input. It takes in the input functions to be a list of strings and a dictionary of variables names and values. It then outputs the gradients and values for each function and variables. It automatically supports both multi-functions/variables.

-- ‘auto_diff’: it is a wrapper for ‘Forward’ and ‘Reverse’ where it automatically detects and uses the more efficient way. It automatically switches to Forward mode when the number of variables are smaller than or equals to the number of functions and switches back to Reverse mode when the number of variables are larger than the number of functions.

apollo_ad/UI.py: it includes a UI function, which is an interactive function that provides prompts to ask users to type in the variables and functions without any coding.

apollo_ad/demo.py： it includes several example use cases of apollo_ad.

4.3. Test suite

Our test suite is in a test file called *test_forward_mode.py* and *test_reverse_mode.py* and are in *apollo_ad*/*tests* folder. We use TravisCI for automatic testing for every push, and Codecov for line coverage metrics. These integrations are 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

```
from .apollo_ad import *
from .UI import UI
from .demo import demo
```

apollo_ad.py

```
class Variable:
    Attributes:
        val: value of the obj unit
        seed: the value of the derivative seed
    Methods:
        __init__(self, value, seed)
        __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)

class Reverse_Mode:
    Attributes:
        val: value of the obj unit
    Methods:
        __init__(self, value)
        gradient(self)
        # Calculate the gradient down the computation graph
recursively
        derivative(self, inputs, seed)
        # Calculate the gradients from the final function with
respect to each input variable.
        __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)

class Forward:
      Attributes:
        val: a dictionary with variable name and values
        fct: a list of function strings
        seed: a dictionary with variable name and its seed, default
is all 1s
     Methods:
        __init__(self, val, fct, seed)

class Reverse:
      Attributes:
        val: a dictionary with variable name and values
        fct: a list of function strings
        seed: a list of function derivative seed, default is all 1s
     Methods:
        __init__(self, val, fct, seed)

class auto_diff:
      Attributes:
        val: a dictionary with variable name and values
        fct: a list of function strings
      seed: a list of function derivative seed or a dictionary with
variable name and its seed, default is all 1s
     Methods:
        __init__(self, val, fct, seed)
```

For the forward mode, the Variable 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. The input is integers or floats for `var` and int/float/list for `der` to the constructor and all the math functions. The list should be input for `der` when there are multiple variables. Note that this class supports multiple variables but with only a single function. For multiple functions, we provide a wrapper called 'Forward' class, where you can pass single/multiple functions. As many functions are provided as static methods, the users may not know when to use the static prefix. Thus, we make a design decision that the input is changed to a string of functions and in the 'Forward' class, we automatically add the static methods prefix to the necessary functions. This allows a hassle-free function input experience.

The reverse mode implementation is discussed below. When the number of variables are smaller than or equals to the number of functions, it is more efficient to use the forward mode and when the number of variables are larger than the number of functions, it is more efficient to use the reverse mode. Thus, we provide a wrapper class called 'auto_diff' that automatically detects and switches to the correct mode.

5. Extension Feature -- Reverse Mode

The main additional feature implemented was 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. Please refer to Section 2 *Background* for a deeper understanding of the required mathematics and concepts related to reverse mode.

Originally, we created the reverse mode as a separate module from the forward mode implementation, but eventually we decided to merge them into an apollo_ad module for readability and package understanding where we had a Reverse() class and a Forward() class, respectively. This helped all of us better understand reverse mode at a very deep level, since it required us to implement the approach from scratch.

We implemented reverse mode by first conceptually thinking about the computational graph and using the reverse mode equation from Section 2.2. It wasn't too hard to calculate the partial derivatives in the forward pass, but the difficulty was in how we wanted to store and use them for the backward pass. Eventually, since we wanted to keep the code structure consistent with how we implemented the forward mode, we decided to store the child nodes for each input and intermediate variable in a list when creating the function. Therefore, when getting the reverse mode derivative for a function at a particular point, we're able to recursively call and update the input and intermediate variables with their respective derivatives with respect to the function using the chain rule, the base case being when we reach a node that already has its derivative calculated. For a more visual understanding please see our YouTube video, and for a more code-driven understanding please see our Reverse_Mode() class, specifically the gradient() and derivative() functions, within the apollo_ad.py script.

Finally, we implemented a UI to reflect that reverse mode typically performs better than forward mode when the number of inputs are greater than the number of outputs. Therefore, whenever a

---

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

user uses the UI class and the number of inputs (variables) is greater than the number of outputs (functions), we'll let the user know that we're using reverse mode on the backend to calculate the derivatives. One of our hopes with this approach is that our package can be used as a learning tool for users to better understand the differences in computational efficiency between forward mode and reverse mode, and when to use each approach to minimize time complexity.

6. Broader Impact

The apollo_ad package is user-friendly, allowing users to utilize the package as a convenient problem-solving tool in automatically calculating derivatives in multiple ways. With this package, users can reach the goal without coding every step, especially when they are dealing with large-scale computation and complex cases. Even users with little experience could operate this package by simply inputting values in our terminal UI. The concepts of forward and reverse mode allow for an ease in being able to differentiate both complex functions and basic functions. At the same time, the range of this package is wide since it allows users to create user-specific functions on their own. The package is impactful due to its ability to easily calculate user-specific functions inheriting instances of variables using the apollo_ad package.

The package would potentially be used within machine learning projects, especially for users making use of automatic differentiation, since the package will also decide whether it is more efficient to use reverse mode or forward mode based on the user's input. This further extension also creates deeper implications in both scientific research and real-word applications.

For education purposes, this package could also be used in higher education or middle school, such as for students who are studying calculus and derivatives. This package could be used like a validation package to see students' accuracy. The package will tell accurate answers without step-by-step calculation, making it fast and efficient. On the other hand, potential negative implications for beginners are that they can avoid the manual work of explicitly calculating derivatives and deeply understanding material on derivatives since this package can be used as a shortcut.

For complex ethical considerations, every method has its two sides. The package is impactful due to its ability to easily calculate user-specific functions. There are also possibilities that user

may utilize this package for negative societal effects, since users don't need to provide information about exactly what they're using apollo_ad for, whether legal or not. Since the potential impact of this package is wide, there may be ethical concerns which are far from our original expectations.

7. Inclusivity Statement

Our project mission is to make apollo_ad a universally accessible, usable, and simple math tool. We are committed to creating a diverse, equal and inclusive workforce. Our software is open source on Github and PyPI for everyone, enabling the creation of pull requests and comments to help improve the package.

Further, we endeavor to build packages that work for everyone by including perspectives from backgrounds that vary by race, ethnicity, religion, gender, age, disability, social backgrounds, sexual orientation, culture, and national origin. Users with any identity can use our packages the way they want, without releasing their personal information. With possible user add-ons, we encourage further contribution on multi-language versions, empowering a more diverse audience and increased participation.

8. Future Additions and Applications

In the immediate next step, we would like to extend our functionality to matrix operations so that users can easily create matrix-like data structures from our package, perform computations and keep track of the gradients. This extension is meaningful because many real world problems may involve hundreds of variables and functions. One of the applications we want to push forward is to use our package to implement a neural network. Neural networks have demonstrated its capability in solving classification and prediction tasks in many areas. One of the most complex and advanced models, BERT[4], has a neural network architecture. The building block of a neural network, a hidden layer, is often represented as a matrix. Each column of the matrix represents the weights of a hidden node and the number of columns corresponds to the output size of this hidden layer. Once we extend our implementation to matrix, we can easily instantiate matrices and build a neural network with multiple hidden layers. We also believe that with the matrix

4 https://arxiv.org/pdf/1810.04805.pdf

operations implemented, our package can be used in an educational environment where it can demonstrate how neural networks work.

```
from apollo_ad import Reverse
var = {'x': 3, 'y': 4}
fct = ['cos(x) + y ** 2', '2 * log(y) - sqrt(x)/3']
out = Reverse(var, fct)
print(out)

# -- Values --
# Function F1: 15.010007503399555
# Function F2: 3.2938507417182654
# -- Gradients --
# Function F1: [-0.14112001  8.        ]
# Function F2: [0.23710829 0.5       ]
```