Team 01 - Milestone 2

Team Members: Sijia (Nancy) Li, Qingyang (Catherine) Ni, Yuqing (Lily) Pan, Jiashu Xu

1.0 Introduction

Differentiation is the operation of finding derivatives of a given function with applications in many disciplines. In classical mechanics, time derivatives are particularly useful in describing physical processes. For example, the first derivative of an object's change in position (displacement) with respect to time is the velocity. The second derivative of an object's displacement is the acceleration. Another famous example is Euler equations, a system of conservation laws (i.e., conservation of mass, momentum, and energy) given by three partial differential equations. In optimization, differentiation can be used to find mimina and maxima of functions. For a single variable, critical or stationary points are points in which the first derivative is zero. In higher dimensions, critical points are points in which the gradient is zero. Another useful technique is the second derivative test, whereby the second derivative of a single variable function or the eigenvalues of the Hessian matrix (a scalar-function's second partial derivatives) in higher dimensions can be used to test for local minimum and local maximum. In deep learning, optimization is used to train neural networks by finding a set of optimal weights that minimizes the loss function.

In particular, there are three techniques for computing derivatives: symbolic differentiation, numerical differentiation, and automatic differentiation. Symbolic differentiation finds derivatives by applying various differentiation rules (e.g., sum rule, constant rule, power rule) to break complex functions into simpler expressions. However, simplification of a complex function or a large number of functions can lead to an exponentially large number of expressions to evaluate. This means that although symbolic differentiation will yield accurate results, it can be too slow and costly to perform such computations in real-life. Numerical differentiation estimates the derivatives from known values of the functions (e.g., finite difference approximation method). However, this technique can lead to inaccurate results due to floating point errors. Automatic differentiation (or algorithmic differentiation) solves these problems. Specifically, automatic differentiation breaks down the original functions into elementary operations (e.g., addition, subtraction, multiplication, division, exponentiation, and trigonometric functions) and then evaluate the chain rule step by step. This technique is able to compute derivatives efficiently (with lower cost than symbolic differentiation) at machine precision (more accurate than numerical differentiation). Since derivatives are ubiquitous in many real-world applications, it is important to have a useful tool to perform differentiation. Natually, the best choice among the three aforementioned methods for computing derivatives is automatic diffferentiation, and `AutoDiff` is a Python package that implements this method.

2.0 Background

2.1 The Chain Rule

The chain rule helps differentiate composition functions. Applying the chain rule, the derivative of a function $f(g(x))$ with respect to a single independent variable x is given by:

$$\frac{df}{dx} = \frac{\partial f}{\partial g}\frac{dg}{dx}$$

For example, given $f(g(x)) = cos(2x)$ and $g(x) = 2x$, then $\frac{\partial f}{\partial g} = -sin(g)$, $\frac{dg}{dx} = 2$, and $\frac{df}{dx} = -2sin(2x)$. For functions with two or more variables, we are interested in the gradient of a function $f(\mathbf{x})$ with respect to all independent variables $\mathbf{x}$, $\nabla_x f$. Applying the chain rule, this is given by:

$$\nabla_{\mathbf{x}} f = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \nabla g_i(\mathbf{x})$$

where g_i's are functions with m variables, $i = 1, 2, \ldots, n$. For example, given $f(g_1(\mathbf{x}), g_2(\mathbf{x})) = sin(g_1) - 2g_2$, and $g_1(\mathbf{x}) = x_1 + 2x_2, g_2(\mathbf{x}) = 3x_1^2 x_2$, then $\nabla g_1(\mathbf{x}) = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \nabla g_2(\mathbf{x}) = \begin{bmatrix} 6x_1x_2 \\ 3x_1^2 \end{bmatrix}$, and $\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1}\nabla g_1(\mathbf{x}) + \frac{\partial f}{\partial g_2}\nabla g_2(\mathbf{x}) = cos(x_1 + 2x_2)\begin{bmatrix} 1 \\ 2 \end{bmatrix} - 2(3x_1^2x_2)\begin{bmatrix} 6x_1x_2 \\ 3x_1^2 \end{bmatrix}$. This is an example with $m = 2$ and $n = 2$.

2.2 Elementary Functions

An elementary function is a combination of elementary operations, and constant, algebraic, exponential, and logarithmic functions as well as their inverses, e.g., $2x, e^x, sin(x), x + 5$. We can decompose a function into smaller parts (elementary functions) in which their symbolic derivatives can be easily computed. The table below shows some examples of elementary functions with their respective derivatives.

Elementary Function	Derivative of the Function
c	0
ax	a
x^2	$2x$
e^x	e^x
$ln(x)$	$\frac{1}{x}$
$sin(x)$	$cos(x)$
$cos(x)$	$-sin(x)$
$tan(x)$	$sec^2(x)$
cf	cf'
x^n	nx^{n-1}
$f + g$	$f' + g'$
$f - g$	$f' - g'$
fg	$fg' + f'g$
f/g	$\frac{f'g - g'f}{g^2}$

2.3 Forward Mode

2.3.1 Forward Primal Trace

Take the function $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ as an example. We develop the forward primal trace by finding intermediate results v_j in which j represents an elementary operation. This is achieved by working from the inside out. Given an arbitrary point (x_1, x_2), we can evaluate the intermediate results at the point. The following table show the forward primal trace of the function $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ evaluated at the point $(\frac{\pi}{6}, 2)$.

Intermediate	Elementary Operation	Numerical Value
$v_{-1} = x_1$	$\frac{\pi}{6}$	0.52359877559

$v_0 = x_2$	2	2
v_1	$3v_{-1}$	1.57079632679
v_2	v_0^3	8
v_3	$sin(v_1)$	1
v_4	$2v_2$	16
v_5	$v_3 + v_4$	17

2.3.2 Computational (Forward) Graph

The computational graph is a way of visualizing the partial ordering of elementary operations with each node representing an intermediate result. The computational graph for the aforementioned function $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ is shown below.

2.3.4 Forward Tangent Trace

Along with the forward primal trace, we also develop the forward tangent trace simultaneously by computing the directional derivative for each intermediate variable $D_p v_j$. By definition, the directional derivative is given by the following equation.

$$D_p v_j = (\nabla v_j)^T p = \sum_{i=1}^{m} \frac{\partial v_j}{\partial x_i} p_i$$

In this definition, p is a m-dimensional seed vector in which m is the number of independent variables. We can specify the derivative of interest using this seed vector. For example, if the goal is to find $\frac{\partial v_j}{\partial x_5}$, then the element p_5 will be one and the remaining elements in the p vector will be zero. The seed vector can be freely selected by the user. Generalizing this, the forward mode automatic differentiation is in essence computing $\nabla f \cdot p$ for a scalar function $f(x)$ and $J \cdot p$ for a vector function $f(x)$ (J is the Jacobian).

The following table shows the forward primal trace and the forward tangent trace evaluated for the seed vectors $p_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $p_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

Forward Primal Trace	Forward Tangent Trace	**Pass** $p^{(j=1)} = [1, 0]^T$	**Pass** $p^{(j=2)} = [0, 1]^T$
$v_{-1} = x_1 = \frac{\pi}{6}$	$D_p v_{-1} = p_1$	$D_p v_{-1} = 1$	$D_p v_{-1} = 0$
$v_0 = x_2 = 2$	$D_p v_0 = p_2$	$D_p v_0 = 0$	$D_p v_0 = 1$
$v_1 = 3v_{-1} = \frac{\pi}{2}$	$D_p v_1 = 3D_p v_{-1}$	$D_p v_1 = 3$	$D_p v_1 = 0$
$v_2 = v_0^3 = 8$	$D_p v_2 = 3v_0^2 D_p v_0$	$D_p v_2 = 0$	$D_p v_2 = 12$
$v_3 = sin(v_1) = 1$	$D_p v_3 = cos(v_1) D_p v_1$	$D_p v_3 = 0$	$D_p v_3 = 0$
$v_4 = 2v_2 = 16$	$D_p v_4 = 2D_p v_2$	$D_p v_4 = 0$	$D_p v_4 = 24$
$v_5 = v_3 + v_4 = 17$	$D_p v_5 = D_p v_3 + D_p v_4$	$D_p v_5 = 0$	$D_p v_5 = 24$

2.3.5 Dual Numbers

By definition, a dual number is given by the equation $z = a + b\epsilon$, where $a, b \in \mathbb{R}$ and ϵ is a very small number not equal to zero such that $\epsilon^2 = 0$. a is the real part and b is the dual part. This structure is very helpful in encoding the primal trace and tangent trace in forward mode automatic differentiation. The primal trace can be encoded by the real part and the tangent trace can be encoded by the dual part, hence the equation $z_j = v_j + D_p v_j \epsilon$.

Going back to the example, we can compute the last intermediate state using dual numbers. The last intermediate state is z_5 which is equal to $z_3 + z_4$, where z_3, z_4, and z_5 are dual numbers. Let $z_3 = a_3 + b_3\epsilon$ and $z_4 = a_4 + b_4\epsilon$, then
$z_5 = (a_3 + b_3\epsilon) + (a_4 + b_4\epsilon) = (a_3 + a_4) + (b_3 + b_4)\epsilon$, where $a_4 + b_4\epsilon$ is the real part and $b_3 + b_4$ is the dual part.

2.4 Reverse Mode

The reverse mode in automatic differentiation is a two-pass process for recovering the partial derivatives $\frac{\partial f_i}{\partial v_{j-m}}$, where f_i's are output functions and v_{j-m} are the intermediate variables. These partial derivatives are called the adjoints. In other words, $\bar{v}_{j-m} = \frac{\partial f_i}{\partial v_{j-m}}$ and $\bar{v}_{j-m}$ is the adjoint of v_{j-m}. The forward pass in the reverse mode will compute the the change in child node v_j with respect to v_j's parent node(s) v_i, denoted as $\frac{\partial v_j}{\partial v_i}$. After that, the reverse pass will build up the chain rule using the formula $\bar{v}_i = \bar{v}_i + \frac{\partial vf}{\partial v_j}\frac{\partial v_j}{\partial v_i} = \bar{v}_i + \bar{v}_j\frac{\partial v_j}{\partial v_i}$, with $\bar{v}_i$ initialized to zero for all i and node j is a child of node i. The last intermediate state is always equal to 1 as the last nodes has no children. The following table illustrates the reverse mode computation for the same example in Section 2.3: $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ evaluated at the point $(\frac{\pi}{6}, 2)$.

Forward Pass		Reverse Pass
Intermediate	**Partial Derivative**	**Adjoint**
$v_{-1} = x_1 = \frac{\pi}{6}$		$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1\frac{\partial v_1}{\partial v_{-1}} = 0$
$v_0 = x_2 = 2$		$\bar{v}_0 = \bar{v}_0 + \bar{v}_2\frac{\partial v_2}{\partial v_0} = 24$
$v_1 = 3v_{-1} = \frac{\pi}{2}$	$\frac{\partial v_1}{\partial v_{-1}} = 3$	$\bar{v}_1 = \bar{v}_1 + \bar{v}_3\frac{\partial v_3}{\partial v_1} = 0$
$v_2 = v_0^3 = 8$	$\frac{\partial v_2}{\partial v_0} = 3v_0^2 = 12$	$\bar{v}_2 = \bar{v}_2 + \bar{v}_4\frac{\partial v_4}{\partial v_2} = 2$
$v_3 = sin(v_1) = 1$	$\frac{\partial v_3}{\partial v_1} = cos(v_1) = 0$	$\bar{v}_3 = \bar{v}_3 + \bar{v}_5\frac{\partial v_5}{\partial v_4} = 1$
$v_4 = 2v_2 = 16$	$\frac{\partial v_4}{\partial v_2} = 2$	$\bar{v}_4 = \bar{v}_4 + \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_4 + \bar{v}_5\frac{\partial v_5}{\partial v_4} = 0 + 1 = 1$
$v_5 = v_3 + v_4 = 17$	$\frac{\partial v_5}{\partial v_3} = 1; \frac{\partial v_5}{\partial v_4} = 1$	$\bar{v}_5 = \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_5} = 1$

3.0 How to Use the `AutoDiff` Package

3.1 Installation

The user can install `AutoDiff` using the following command:

```
python3 -m pip install --upgrade pip
python3 -m pip install -U pytest pytest-cov
python3 -m pip install -r requirements.txt
```

This will be available in the future: `python3 -m pip install AutoDiff`

Currently, the user can use the `AutoDiff` package by directly cloning from the package's GitHub repository using one of the two commands below:

```
git clone https://code.harvard.edu/CS107/team01.git
git clone git@code.harvard.edu:CS107/team01.git
```

3.2 Usage

Please view the full documentation here. The following code snippet demonstrates how users will interact with our package.

3.2.1 Example 1: One Independent Variable

```
>>> from AutoDiff import Forward, Reverse
>>> import numpy as np
>>> # Create a node x with value 5.
>>> x = 5
>>> # Create a function y = exp(cos(x)+2).
>>> f = lambda x: np.exp(np.cos(x)+2)
>>> # Create a forward mode instance.
>>> g = Forward(f, x)
>>> # Evaluate the value of y at x=5
>>> g.val
9.812550066983217
>>> # Calculate dy/dx = -exp(2+cos(x))sin(x) for x = 5.
>>> g.der
array([9.40949246])
>>> # Create a reverse mode instance.
>>> rev = Reverse(f, x)
>>> # Evaluate the value of y at x=5
>>> rev.val
9.812550066983217
>>> # Calculate dy/dx = -exp(2+cos(x))sin(x) for x = 5.
>>> rev.der
array([9.40949246])
```

3.2.2 Example 2: Two Independent Variables

```
>>> from AutoDiff import Forward, Reverse
>>> import numpy as np
>>> # Create two nodes x1 with value 5 and x2 with value 3.
>>> x1 = 5
>>> x2 = 3
>>> # Create a function y = exp(cos(x1)+2sin(x2)).
>>> def f(x1, x2):
        return np.exp(np.cos(x1)+2*np.cos(x2))
>>> # Create a forward mode instance.
>>> g = Forward(f, x1, x2)
>>> # Evaluate the value of y at x1 = 5, x2 = 3
>>> g.val
0.1833565231089554
>>> # Calculate dy/dx at x1 = 5, x2 = 3
>>> g.der
array([0.17582502, -0.05175055])
>>> # Create a reverse mode instance.
>>> rev = Reverse(f, x1, x2)
>>> # Evaluate the value of y at x1 = 5, x2 = 3
>>> rev.val
0.1833565231089554
>>> # Calculate dy/dx at x1 = 5, x2 = 3
>>> rev.der
array([0.17582502, -0.05175055])
```

3.2.3 Example 3: Newton's Method

Assume we want to compute $\sqrt{2}$, that is, finding x such that $f(x) \triangleq x^2 - 2 = 0$. Let's begin with a random guess $x_0 = 1.4$. Newton method essentially compute

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}$$

```
>>> from AutoDiff import Forward
>>> import numpy as np
>>> f = lambda x: x**2 - 2
>>> x0 = 1.4
>>> def newton(f, x0, tol=1e-10):
        if abs(f(x0)) < tol:
            return x0
        g = Forward(f, x0)
        new_x0 = x0 - g.val / g.der
        return newton(f, new_x0)
>>> our_sol = newton(f, x0)
>>> our_sol
array([1.41421356])
>>> assert np.allclose(np.sqrt(2), our_sol)
```

4.0 Software Organization

This section briefly discusses our plan on organizing the Python package `AutoDiff`. The directory structure is shown below, following the recommended python package structure.

```
. # root dir
├── docs # development documentations
│   ├── milestone1.pdf
│   └── milestone2_progress.md
│   └── milestone2.pdf
│   ├── src # this folder holds all the source code for
milestone report
│   │   ├── milestone1.ipynb
│   │   └── milestone2.ipynb
│   └── source # autogenerated docs by sphinx
│       └── Makefile
├── LICENSE
├── README.md
├── setup.cfg
├── setup.py
├── requirements.txt
├── test
│   ├── test_node.py # test for node module
│   ├── test_forward.py # test for forward module
│   └── test_rnode.py # test for rnode module
│   └── test_reverse.py # test for reverse module
└── AutoDiff # our library source code
    ├── __init__.py
    ├── node.py # node module
    ├── forward.py # forward module
    ├── rnode.py # rnode module
    └── reverse.py # reverse module
```

The functionalities of each aforementioned directory/file is shown below.

- `docs` : This directory contains the major deliverables from each milestone.
- `docs/source` : We use sphinx for automated documentation generation due to its popularity. Such flexible automated tool read content from the docstring and create documentations automatically.
- `examples` : This directory provides examples to show users how to use the package.
- `LICENSE` : This is the file specifying the license for our python package.
- `README.md` : This is the readme for the project repository.
- `setup.cfg` : This file contains the setup configuration and will be used by setuptools.

- `setup.py` : This is used for defining and publishing our package. It contains the version number, dependencies, upstream GitHub URL, and etc. The functionality is described in here.
- `requirements.txt` : This is the basic requirements needed for user to use our package.
- `test` : This is the directory that contains the pytest modules. They are unit tests. We plan to separate these tests into modules. We plan to use CodeCov to show the test coverage for our code.
- `AutoDiff` : This is the folder for the actual source code. We follow the recommended structure of python module shown in the python documentation.

To clearly visualize the directory structure, we did not include files such as `.gitignore` , `.github` for github action, `.codecov.yml` for CodeDev, and `requirements.txt` for listing dependencies in local development.

For distribution, we package the code by the standard setuptools and then distribute our software using Python Package Index (PyPi). In this way, future users can download the package by simply using the following command.

```
pip install AutoDiff
```

GitHub action is used to conduct CI with pytest. Each time a commit is pushed to GitHub, a pytest test is initiated. Moreover, the traditional Git workflow is followed. The main branch is used for stable release of the package and all development is conducted in separate branches. We only merge to main branch if the code in development branch passes the test. Finally, CodeCov is used to show the test coverage of the package.

5.0 Implementation

The `AutoDiff` package currently contains two classes: `Node` and `Forward` . Our implementation currently needs to depend on the NumPy library.

Before computing the gradient of the input function, the input function is wrapped into a graph structure that stores the partial ordering of the intermediate results v_j (Section 2.3). To realize the graph structure, we need to implement the class `Node` , where each node contains a reference of its parents. Since we have constructed the core data structure `Node` of the input function, we can implement the forward mode of automatic differentiation in the class `Forward` to compute the Jacobian of the input function. The following subsections are more detailed introduction of the classes with class attributes and methods.

5.1 `class Node`

The class `Node` turns the function into the core data structure -- graph. Attributes of `Node` includes `self.val`, `self.der`, `self.parent`, `self.op`. The class `Node` incorporate the structure of dual numbers (Section 2.3.5), where `self.val + self.der` ε is assemble to the dual number's structure $a + b\varepsilon$. In this class, we further overload the elementary operators and useful numpy functions.

```
import numpy as np

class Node:
    """
    This is a class that implements the Dual numbers and dunder
methods to overload
    built-in operators including negation, addition, subtraction,
multiplication, true
    division, and power. The class also overloads numpy operators
including square root,
    exponential, logarithm, sine, cosine, and tangent. Reflective
operations are also
    included.

    :param val: The value of the Node, it can be an interger or float
or a 1D numpy array
    for 1D problem, or a multi-dimensional numpy array for multi-
dimensional problem
    :type val: integer or float or numpy array
    :param der: The derivative of the Node, defaults to 1. It can be
an interger or float
    or a 1D numpy array for 1D problem, or a multi-dimensional numpy
array for
    multi-dimensional problem
    :type der: integer or float or numpy array
    :param parent: A list of parent Nodes of the current Node
    :type parent: list
    :param op: A list of strings representing operations
    :type op: list
    """

    def __init__(self, value, derivative=1):
        self.val = value
        self.der = derivative
        self.parent = []
        self.op = []

    def update_node(self, parent, op):
        self.parent = parent
        self.op = op
        return self

    """
    Some examples of elementary operators that are overloaded
```

```
    """
    def __neg__(self):
        value = -self.val
        derivative = -1 * self.der
        return Node(value, derivative).update_node([self], ['-1*'])

    def __add__(self, other):
        if isinstance(other, Node):
            value = self.val + other.val
            derivative = self.der + other.der
            return Node(value, derivative).update_node([self, other],
['+'])
        elif not isinstance(other, self._supported_types):
            raise TypeError(f"Type `{type(other)}` is not supported
for addition")
        else:
            value = self.val + other
            derivative = self.der
            return Node(value, derivative).update_node([self], ['+',
other])

    def __sub__(self, other):
        if isinstance(other, Node):
            value = self.val - other.val
            derivative = self.der - other.der
            return Node(value, derivative).update_node([self, other],
['-'])
        elif not isinstance(other, self._supported_types):
            raise TypeError(f"Type `{type(other)}` is not supported
for subtraction")
        else:
            value = self.val - other
            derivative = self.der
            return Node(value, derivative).update_node([self], ['-',
other])

    def __mul__(self, other):
        if isinstance(other, Node):
            value = self.val * other.val
            derivative = self.der * other.val + other.der * self.val
            return Node(value, derivative).update_node([self, other],
['*'])
        elif not isinstance(other, self._supported_types):
            raise TypeError(f"Type `{type(other)}` is not supported
for multiplication")
        else:
            value = self.val * other
            derivative = self.der * other
            return Node(value, derivative).update_node([self], ['*',
other])
```

```
    def sqrt(self):
        value = np.sqrt(self.val)
        derivative = 0.5/np.sqrt(self.val) * self.der
        return Node(value, derivative).update_node([self],
['sqrt()'])

    def log(self):
       value = np.log(self.val)
        derivative = 1 / self.val * self.der
        return Node(value, derivative).update_node([self], ['log()'])

    """
    Other elementory operators and reflective operators that are also
overloaded
    """
    __truediv__(self, other: Node)
    __pow__(self, )
    exp(self)
    sin(self)
    cos(self)
    tan(self)
    __radd__(self, other)
    __rsub__(self, other)
    __rmul__(self, other)
    __rtruediv__(self, other)
```

5.2 class Forward

In the class `Forward`, we need to model arbitrary high-level function f. We treat a vector function $f: \mathbb{R}^m \mapsto \mathbb{R}^n$ as a list of scalar functions $f: \mathbb{R}^m \mapsto \mathbb{R}$. Our key observation is that, once the number of input variables are known, we can iterate over all natural basis and can obtain the jacobian in one pass.

Specifically, the `grad` method below ompute the full Jacobian by looping through the vector of scalar functions.

```
from node import Node
class Forward:
    """
    Forward mode AD for vector functions of vectors
    """

    def __init__(self, f: callable, *variables):
        """
        Initialize forward class. For detailed implementation see
:py:meth:`AutoDiff.forward.Forward.grad`.

        Args:
```

```
            f: Function that is callable

            variables: inputs
        """
        self.val, self.der = self.grad(f, *variables)

    @staticmethod
    def grad(f: callable, *variables):
        """
        Evaluate the full Jacobian in forward mode. This is the
method that is used internally
        by :py:meth:`AutoDiff.forward.Forward.__init__`.

        For each (scalar or multivariate) function ``f``,
        use :math:`m` passes with different seed vector
:math:`\mathbf{e}`,
        where each natural basis :math:`\mathbf{e} \in
\mathbb{R}^{m}`, and :math:`m` is the number in ``variables``.

        :param f: callable
        :param variables: input
        :return: Jacobian
            Stack the gradient rows into the full Jacobian.
        """
        def natural_basis(N, i):
            return np.eye(N)[i]
        num_variables = len(variables)
        variables = [
            Node(var, derivative=natural_basis(num_variables, i))
            for i, var in enumerate(variables)
        ]
        output = f(*variables)
        return output.val, output.der
```

6. Future Features

For future features, we will implement multi-variate function functionality for `class Forward` and the comparison operator overloading in `class Node`. In addition, we will implement the Reverse Mode of Automatic Differentiation by adding two new classes: `class RNode` and `class Reverse`. Furthermore, we will extend the package for various optimizers as well as creating computational graph visualization. Lastly, the `AutoDiff` package will be published on PyPI and some driver scripts will also be provided to users for ease of use of the package.

6.1 Multi-variate Function Functionality for `class Forward`

We will implement the multivariate function functionality to accept $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ and perform further testing on the `class Forward`.

6.2 Comparison for `class Node`

We will implement the comparison between the values of two Node instances by overloading the functions `__gt__`, `__lt__`, `__gt__`, `__lt__`, and `__eq__`.

6.3 Reverse Mode

6.3.1 `class RNode`

In the class `RNode`, we implement the basic Reverse Node class for performing Reverse mode. The parameters `self.val` stores the function value and calculates the Forward Pass. The parameter `self.parent` stores a list of parents of the node. We then calculate the gradient of the function using `self.val` and store it in `self.der`. We will achieve the multivariable functionality in class `Reverse`.

```
import numpy as np
class RNode:
    """
    Reverse mode AD for vector functions of vectors
    """
    def __init__(self, val):
        self.val = val
        self.der = None
        self.parent = []

    def grad(self):
        if self.der is None: # basic variable
            self.der = sum(p.val * p.grad() for p in parent)
        return self.der


    def __neg__(self):
        rnode = RNode(-self.val)
        self.parent.append((-1, rnode))
        return rnode

    def __add__(self, other):
        if isinstance(other, RNode):
            rnode = RNode(self.val + other.val)
            self.parent.append((1., rnode))
            other.parent.append((1., rnode))
            return rnode
        elif not isinstance(other, self._supported_types):
            raise TypeError(f"Type `{type(other)}` is not supported
for addition")
        else:
            rnode = RNode(self.val + other)
            self.parent.append((1., rnode))
```

```
            return rnode

    def __sub__(self, other):
        if isinstance(other, RNode):
            rnode = RNode(self.val - other.val)
            self.parent.append((1., rnode))
            other.parent.append((-1., rnode))
            return rnode
        elif not isinstance(other, self._supported_types):
            raise TypeError(f"Type `{type(other)}` is not supported
for subtraction")
        else:
            rnode = RNode(self.val - other)
            self.parent.append((1., rnode))
            return rnode

    def __mul__(self, other):
        if isinstance(other, RNode):
            rnode = RNode(self.val * other.val)
            self.parent.append((other.val, rnode))
            other.parent.append((self.val, rnode))
            return rnode
        elif not isinstance(other, self._supported_types):
            raise TypeError(f"Type `{type(other)}` is not supported
for multiplication")
        else:
            rnode = RNode(self.val * other)
            self.parent.append((other, rnode))
            return rnode

    """
    Other elementory operators and reflective operators also need to
be overloaded
    """
    __truediv__(self, other)
    sqrt(self)
    exp(self)
    log(self)
    sin(self)
    cos(self)
    tan(self)
    __pow__(self, other)
    __radd__(self, other)
    __rsub__(self, other)
    __rmul__(self, other)
    __rtruediv__(self, other)
```

6.3.2 class Reverse

In the class `Reverse`, we perform the reverse mode automatic differentiation. The `grad()` method evaluates the function at user-given values of x and compute the full Jacobian by looping through the vector of scalar functions, notice that by our implementation of the `RNode` class, we get the partial derivatives column in forward pass of the reverse mode computation table while initializing functions.

```
from rnode import RNode
class Reverse:
    def __init__(self, f_list: callable, *variables):
        """
        Initialize Reverse class.
        Args:
            f: Function that is callable
            variables: inputs
        """
        self.val, self.der = self.grad(f_list, *variables)

    @staticmethod
    def grad(f_list: callable, *variables):
        """
        Evaluate Jacobian in reverse mode.
        For each scalar function f in the f_list,
        use a forward pass and a reverse pass.
        Stack the gradient rows into the full Jacobian.
        """
        variables = [Node(var, None) for var in variables]
        f_grad = []
        f_value = []
        for f_cur in self.f_list:
            # Evaluate gradient for each scalar funtion
            for var in variables:
                var.clear()
            f_value.append(f_cur(*variables))
            f_grad.append[var.grad() for var in variables]
        return np.array(f_value), np.array(f_grad)
```

6.4 Optimization

We will provide optimizers including Netwon's method, secant method, and SGD.

- Newton's method iteratively update

$$x_{k+1} \leftarrow x_k - f(x_k)/f'(x_k)$$

- SGD also iterative the following

$$x_{k+1} \leftarrow x_k - \eta \nabla_x f(x_k)$$

6.5 Computational Graph Visualization

There are two options for creating a visualization for the computational graph of an input function. The first option is to create a static computational graph visualization using the open source `Graphviz` software. The second option is to save the data in a `data.csv` file with the following data: variable name, value, derivative, parent, and operation. This data will then be used to create an interactive visualization of the computational graph in which hovering over each node will show the node's corresponding value and derivative. The aim is to implement the static computational graph, if time allows, we will make the visualization interactive.

6.6 Publishing on PyPI

We will follow the tutorial here, and publish our code on PyPI so that user can use our pacakge by `pip install AutoDiff` .

6.7 Driver Script For Users

We will provide some sample driver scripts for users, for example, Newton's Method.

7.0 License

We chose to use the MIT License. As described by the MIT license, it will allow users to reuse the code for commercial or private use, distribution and modification (essentially any purpose), as long as the users include the original copy of the MIT license in their distribution. The MIT license is a permissive license as it does not require the user to make their work publicly available as well. Automatic differentiation is a project that has been worked on by many users, i.e. the software is not very substantial, and the project can be very useful for anyone wants to differentiate their functions quickly. Since this is a course project, we do not want to profit from the software, but we want to use the software as a component of a broader service. Therefore we do not feel the need to limit the use of our software, and it is not necessary for us to use a copyleft license to force out user to make their project open source. Due to similar reasons, we do not care about patents.