Team 01 - Documentation

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 minima 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. Naturally, the best choice among the three aforementioned methods for computing derivatives is automatic differentiation, and `AutoDiff` is a Python package that implements this method.

Three additional features are also implemented in the `AutoDiff` package: Reverse Mode, Optimization, and Computation Graph Visualization. The Reverse Mode for automatic differentiation is a two-pass process for recovering the partial derivatives. Optimization is a key concept in many fields (e.g., engineering, finance) and it helps us find the best possible solutions to a wide range of problems. In particular, Newton's Method and Stochastic Gradient Descent (SGD) are implemented. Finally, the `graphviz` software is used to develop the computation graph given an input function.

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 $\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1} \nabla g_1(\mathbf{x}) + \frac{\partial f}{\partial g_2} \nabla g_2(\mathbf{x}) = cos(x_1 + 2x_2) \begin{bmatrix} 1 \\ 2 \end{bmatrix} - 2(3x_1^2 x_2) \begin{bmatrix} 6x_1x_2 \\ 3x_1^2 \end{bmatrix}$. 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

Elementary Function	Derivative of the Function
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

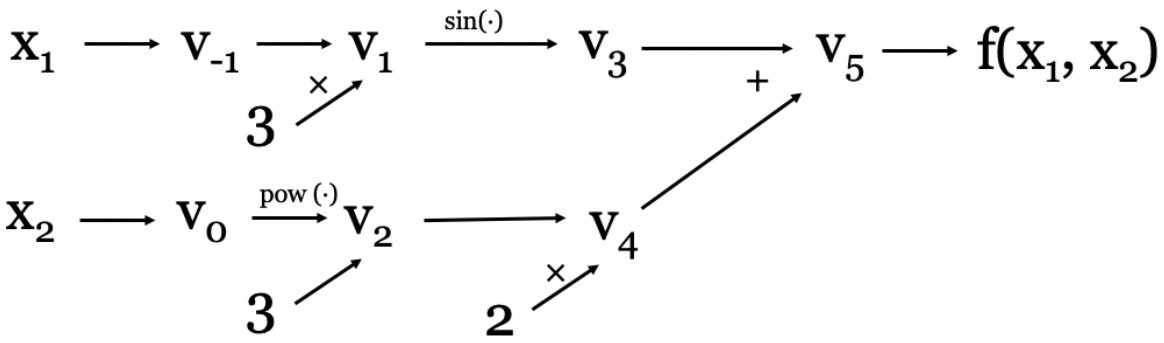
2.3.1 Forward Primal Trace

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.



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

2.3.4 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
$\bar{v}_i = \bar{v}_i + \frac{\partial v_f}{\partial v_j}\frac{\partial v_j}{\partial v_i} = \bar{v}_i + \bar{v}_j\frac{\partial v_j}{\partial v_i}$, 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**
$v_{-1} = x_1$ $= \frac{\pi}{6}$		$\bar{v}_{-1} = \bar{v}_{-1}$ $+ \bar{v}_1\frac{\partial v_1}{\partial v_{-1}}$ $= 0$
$v_0 = x_2 = 2$		$\bar{v}_0 = \bar{v}_0$ $+ \bar{v}_2\frac{\partial v_2}{\partial v_0}$ $= 24$
$v_1 = 3v_{-1}$ $= \frac{\pi}{2}$	$\frac{\partial v_1}{\partial v_{-1}} = 3$	$\bar{v}_1 = \bar{v}_1$ $+ \bar{v}_3\frac{\partial v_3}{\partial v_1} = 0$
$v_2 = v_0^3 = 8$	$\frac{\partial v_2}{\partial v_0} = 3v_0^2 = 12$	$\bar{v}_2 = \bar{v}_2$ $+ \bar{v}_4\frac{\partial v_4}{\partial v_2} = 2$
$v_3 = sin(v_1)$ $= 1$	$\frac{\partial v_3}{\partial v_1} = cos(v_1)$ $= 0$	$\bar{v}_3 = \bar{v}_3$ $+ \bar{v}_5\frac{\partial v_5}{\partial v_4} = 1$

Forward Pass		Reverse Pass
$v_4 = 2v_2$ $= 16$	$\frac{\partial v_4}{\partial v_2} = 2$	$\bar{v}_4 = \bar{v}_4$ $+ \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_4}$ $= \bar{v}_4$ $+ \bar{v}_5\frac{\partial v_5}{\partial v_4} = 0$ $+ 1 = 1$
$v_5 = v_3 + v_4$ $= 17$	$\frac{\partial v_5}{\partial v_3} = 1; \frac{\partial v_5}{\partial v_4} = 1$	$\bar{v}_5 = \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_5}$ $= 1$

2.5 Optimization

2.5.1 Newton's Method

Newton's Method is an algorithm that computes an approximate solution x^* to the equation $f(x) = 0$, given that the function f is differentiable. The algorithm runs as follows.

Algorithm (Newton's Method):
Input: initial guess x_0, input function f, maximum iteration max_iter, tolerance tol
Output: minimum or maximum of function f

while $k < max\_iter$ do
  if $f(x_k) < tol$ then
    x^* = x_k
    return x^*
  end

  if scalar function then
    $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$
  else
    $J_f(x_k)\Delta x_k = -F(x_k)$
    $x_{k+1} \leftarrow x_k + \Delta x_k$
  end
  $k = k + 1$
end

2.5.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an algorithm for optimizing an objective function by iteratively descending in the negative direction of the gradient. The algorithm runs as follows.

Algorithm (SGD):
Input: initial guess x_0, input function f, learning rate η, maximum iteration max_iter,

tolerance tol
Output: minimum or maximum of function f

while $k < max\_iter$ do
  if $f(x_k) < tol$ then
    x^* = x_k
    return x^*
  end
  $x_{k+1} \leftarrow x_k - \eta \Delta f(x_k)$
  $k = k + 1$
end

2.6 The `graphviz` Software

`graphviz` is an open-source visualization software for representing graphs and networks. The computation graph of functions can be seen as a graph in which all the variables are nodes on the graph and the connections between variables are edges on the graph. The elementary operations can be seen as edge names in the graph. The documentation for the `graphviz` software can be found here.

3.0 How to Use the `AutoDiff` Package

3.1 Installation

3.1.1 User Installation Guide

3.1.1.1 Manual Installation

We publish `AutoDiff` on the testPyPI, and user can simply install `AutoDiff` and its dependency by the following command. Note that the name `AutoDiff` has already been taken on PyPI, therefore, the name `AutoDiff-Team01` is used instead.

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-
url https://pypi.org/simple AutoDiff-Team01
```

We have additional feature to visualize computation graph. To use this functionality user can install graphviz here and install python wrapper by the following command

```
pip install graphviz
```

3.1.1.2 Installation Using Custom Docker

Alternatively, users can skip the above installation (for example if graphviz is not available in the machine). Users can download the Docker file named `Dockerfile` in our repo. Inside the directoy that contains the `Dockerfile`, users can build docker container, demonstrated by the following commands

```
# go to the directory that contains our Dockerfile
cd <directory with Dockerfile>
docker build .
# for example, if the above command ends with
#    Successfully built d8b31c5835d6
# the container id should be d8b31c5835d6
docker run -it <container id>
# then you can use our package in docker!
```

3.1.1.3 Installation Using Our Docker

We also build the container and provide the built container that contains all depdendencies and AutoDiff on docker hub: 13052423200/autodiff. Please run

```
docker pull 13052423200/autodiff
# then you can use our package in docker!
```

3.1.2 Developer Installation Guide

Developers can git clone our repo and install from the source using the following command

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

Then excute the following command

```
cd team01
# for main functionalty
pip install .
```

If developers would like to explore additional features such as computational graph plotting, as well as testing:

```
# for all features including test and plot
pip install .[all]
```

Developers can also build or pull docker, please see user Installation guide above.

3.2 Usage

Please see examples of how to use our package in the `example` folder. Forward Mode AD and Reverse Mode AD examples are given in `forward_mode.py` and `reverse_mode.py` respectively. Computation graph example is given in `plot_graph.py`. And driver code for optimization methods is provided in `newton.py` and `sgd.py` corresponding to Newton's Method and Stochastic Gradient Descent respectively.

For more examples, please refer to `docs/documentation.pdf` section **3.0 How to Use the `AutoDiff` Package**, or view the full documentation with examples here.

For developer, checkout `test` folder for more examples and usages.

The following sample code snippet demonstrates how users can interact with our package.

3.2.1 Example 1: univariate scalar function

```
>>> 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: multivariate scalar function

```
>>> from AutoDiff import Forward, Reverse
>>> import numpy as np
>>> # Create two nodes x1 with value 5 and x2 with value 3.
>>> x = [5, 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, *x)
>>> # 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, *x)
>>> # 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: univariate vector function

```
>>> from AutoDiff import Forward, Reverse
>>> import numpy as np
>>> x = 50
```

```
>>> def f(x):
>>>     return [np.sin(x), np.cos(x)]
>>> g = Forward(f, x)
>>> g.val
array([-0.26237485,  0.96496603])
>>> g.der
array([0.96496603, 0.26237485])
>>> rev = Reverse(f, x)
>>> rev.val
array([-0.26237485,  0.96496603])
>>> rev.der
array([[0.96496603, 0.26237485]])
```

3.2.4 Example 4: multivariate vector function

```
>>> from AutoDiff import Forward, Reverse
>>> import numpy as np
>>> x = [5, 3]
>>> def f(x1, x2):
>>>     return [np.exp(np.cos(x1)+2*np.cos(x2)), x1 * x2, 2]
>>> g = Forward(f, *x)
>>> g.val
array([ 0.18335652, 15.        ,  2.        ])
>>> g.der
array([[ 0.17582502, -0.05175055],
       [ 3.        ,  5.        ],
       [ 0.        ,  0.        ]])
>>> rev = Reverse(f, *x)
>>> rev.val
array([ 0.18335652, 15.        ,  2.        ])
>>> rev.der
array([[ 0.17582502, -0.05175055],
       [ 3.        ,  5.        ],
       [ 0.        ,  0.        ]])
```

3.2.5 Example 5: 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
├── .github/workflows # this folder contains all Github
workflow actions yml files
|   ├── coverage.yml
|   ├── sphinx_build.yml
|   └── test.yml
├── docs # development documentations
|   ├── documentation.pdf
|   ├── milestone1.pdf
|   ├── milestone2_progress.md
|   └── milestone2.pdf
|   ├── src # this folder holds all the source code for
milestone report
|   |   ├── documentation.ipynb
|   |   ├── milestone1.ipynb
|   |   └── milestone2.ipynb
|   └── source # autogenerated docs by sphinx
|       └── Makefile
├── example # examples for how to use this package
|   ├── forward_mode.py
|   ├── newton.py
|   ├── plot_graph.py
|   ├── reverse_mode.py
|   └── sgd.py
├── LICENSE
├── README.md
├── Dockerfile
├── setup.py
├── requirements.txt
├── test
|   ├── __init__.py
|   ├── run_coverage.sh # shell script that print out the
coverage report for code base
|   ├── run_test.sh # shell script that run all tests for code
base
|   ├── test_node.py # test for node module
|   ├── test_forward.py # test for forward module
|   ├── test_optimization.py # test for optimization modules
|   ├── test_plot_computation_graph.py # test for
plot_computation_graph module
|   └── test_rnode.py # test for rnode module
```

```
│   └── test_reverse.py # test for reverse module
└── AutoDiff # our library source code
    ├── graphvis # directory for computation graph
visualization additional feature
    │   ├── __init__.py
    │   ├── computationGraph # automatically generated file
from graphviz (an example)
    │   ├── computationGraph.png # automatically generated
computation graph image (an example)
    │   └── plot_computation_graph.py # plot_computation_graph
module
    ├── optim # directory for optimization additional feature
    │   ├── __init__.py
    │   ├── newton.py # Newton's method module
    │   └── sgd.py # SGD module
    ├── __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.

- `.github/workflows` : This directory contains files for github actions for continuous integration
- `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.
- `example` : 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.
- `Dockerfile` : This is the docker file that contains the configurations of our docker image.
- `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 and these tests are separated into modules. Two shell scripts are provided, all tests in the

`test` folder will be run by executing `run_test.sh`, and a coverage report will be produced by executing `run_coverage.sh`.

- `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, `.vscode/settings.json` file for PyTest on Visual Studio Code,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, users can download the package by simply using the following command.

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-
url https://pypi.org/simple AutoDiff-Team01
```

Tests for the source code in `AutoDiff` package are stored in the `test` directory. These are `pytest` tests. Since the development of the `AutoDiff` package is done using Visual Studio Code (VSCode), the testing functionality in VSCode is leveraged by specifying the location of the tests and the type of tests (i.e., `pytest`) in the `settings.json` file in the `.vscode` directory. Developers can go into the "Testing" tab in VSCode and run any test of interest. VSCode also creates a button besides all tests in the `test` folder, so that all tests can be run individually.

In addition, GitHub action is used to conduct continuous integration (CI), including running `pytest` tests. Each time a commit is pushed to GitHub, `pytest` tests are initiated. Specifically, in the `.github/workflows` directory, the `test.yml` file calls the `run_test.sh` shell script and the list of test cases specified in `tests` are run. The `coverage.yml` file checks the test coverage for the source code in the `AutoDiff` package by calling the `run_coverage.sh` shell script and a test coverage report will be generated. Notice that there is another file named `sphinx_build.yml`. This is for automatically generating documentation for the `AutoDiff` package. Locally, `run_test.sh` and `run_coverage.sh` are the two shell scripts that can be edited and run to conduct testing on the source code and check test coverage.

Finally, the traditional Git workflow is followed. The main branch is used for stable release of the package and all development and testinf is conducted in separate branches. Developers only merge to main branch if the code in development branch passes the test. The test coverage report is generated by `pytest` and published on the GitHub page here.

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`, and `self.v_index`. 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 $a + b\epsilon$. 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
    """
```

```
    _supported_types = (int, float)
    v_index = 0

    def __init__(self, value, derivative=1):
        """Constructor method
        """
        self.val = value
        self.der = derivative
        self.parent = []
        self.op = []
        type(self).v_index += 1
        self.v_index = f'v{type(self).v_index}'

    def update_node(self, parent, op):
        """Update a list of parent Nodes of the current Node with
their operations.
        :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

        :return: Returns the current Node with parents and operations
updated
        :rtype: Node
        """
        self.parent = parent
        self.op = op
        return self

    def __str__(self):
        return f'Node: vindex={self.v_index}, val={self.val}, der=
{self.der}, parent={self.parent}, and op={self.op}.'

    def __repr__(self):
        return f'A Node object with index of {self.v_index}, value of
{self.val}, derivative of {self.der}, parent of {self.parent}, and
operator of {self.op}.'

    """
    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):
        if self.val < 0:
            raise ValueError('Cannot take square root of negative
number.')
        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, base=np.e):
        if self.val <= 0:
            raise ValueError('Cannot take the log of a negative
number.')
        if base == np.e:
            value = np.log(self.val)
```

```
            derivative = 1 / self.val * self.der
            return Node(value, derivative).update_node([self],
['log()'])
        else:
            value = np.log(self.val) / np.log(base)
            derivative = 1 / (self.val * np.log(base)) * self.der
            return Node(value, derivative).update_node([self],
[f'log{base}()'])

    """
    Other elementory operators and reflective operators that are also
overloaded
    """
    logistic(self)
    __truediv__(self, other)
    __pow__(self, other)
    exp(self)
    sin(self)
    cos(self)
    tan(self)
    arcsin(self)
    arccos(self)
    arctan(self)
    sinh(self)
    cosh(self)
    tanh(self)
    __radd__(self, other)
    __rsub__(self, other)
    __rmul__(self, other)
    __rtruediv__(self, other)
    __rpow__(self, other)
    __lt__(self, other)
    __gt__(self, other)
    __le__(self, other)
    __ge__(self, other)
    __eq__(self, other)
    __ne__(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.

```
import numpy as np
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.output = self.grad(f, *variables)

    @staticmethod
    def grad(f: callable, *variables):
        r"""
        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: A callable function object to perform
differentiation on
        :type f: function object
        :param variables: The input for variables of function ``f``
        :type variables: integer or float or numpy array or list of
intergers or floats

        :return: function evaluation at variable x, Jacobian, a
single output node (scalar function) or a list of output nodes
(multivariate) function)
            Stack the gradient rows into the full Jacobian.
        :rtype: Node class object or list of Node class objects
        """
        # Helper function when Forward class is initialized, see
usage in init.
        num_variables = len(variables)
        # initialize the intermediate result index
        Node.v_index = -num_variables
        # Convert variables into Nodes and store in a list
        variables = [
            Node(var, derivative = np.eye(num_variables)[i])
            for i, var in enumerate(variables)
        ]
        # Perform the forward mode
        output = f(*variables)
```

```
        if isinstance(output, list): # for vector functions (a list
of outputs)
            output = [o if isinstance(o, Node) else Node(o,
np.zeros(num_variables)) for o in output]
            values = np.array([o.val for o in output])
            ders = np.stack([o.der if len(o.der) > 1 else o.der[0]
for o in output])
            return values, ders, output
        else: # for scalar functions (a single output)
            if not isinstance(output, Node):
                output = Node(output, np.zeros(num_variables))
            return output.val, output.der if len(output.der) > 1 else
output.der[0], output
```

6. Additional Features and Extensions

For additional features and extensions, the Reverse Mode of Automatic Differentiation is achieved by two new classes: `class RNode` and `class Reverse`. We also extend the package for various optimizers as well as creating computational graph visualization.

6.1 Reverse Mode

6.1.1 `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 tuples, containing the partial derivative of parent with respect to self and the parent of the node. We then calculate the gradient of the function and store it in `self.der`. We achieve the multivariable functionality in class `Reverse`.

```
import numpy as np
class RNode:
    """This is a class that implements the Reverse Nodes for reverse
mode calculation of
    automatic differentiation 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 RNode, 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 parent of the RNode, defaults to None. 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 RNodes of the current RNode
    :type parent: list
    """
    def __init__(self, val):
        """Constructor method
        """
        self.val = val
        self.der = None
        self.parent = []

    def clear(self):
        """Clears self's and all self's parent's derivative field
        """
        self.der = None
        for _, p in self.parent:
            p.clear()

    def grad_vec(self, output_depend):
        """Helper function for Reverse AD, produces all gradient for
its parents
        and any parents defined in the intermediate step. See usage
in Reverse class.
        """
        gradient = []
        for i in range(len(self.parent)):
            gradient.append(self.parent[i][0] * self.parent[i]
[1].grad(output_depend))
        return gradient

    def grad(self, output_depend):
        """Helper function for grad_vec(), see usage in grad().
        """
        if self.parent == []:
            self.der = 1.0
            output_depend.append(self)
        else:
            self.der = sum(p[0] * p[1].grad(output_depend) for p in
self.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
    """
    __str__(self)
    __repr__(self)
    __truediv__(self, other)
    __pow__(self, other)
    sqrt(self)
    log(self)
    logistic(self)
    exp(self)
    sin(self)
    cos(self)
```

```
    tan(self)
    arcsin(self)
    arccos(self)
    arctan(self)
    sinh(self)
    cosh(self)
    tanh(self)
    __radd__(self, other)
    __rsub__(self, other)
    __rmul__(self, other)
    __rtruediv__(self, other)
    __rpow__(self, other)
    __lt__(self, other)
    __gt__(self, other)
    __le__(self, other)
    __ge__(self, other)
    __eq__(self, other)
    __ne__(self, other)
```

6.1.2 `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 computes the full Jacobian at x, 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 the function.

```
import numpy as np
from .rnode import RNode
class Reverse:
    def __init__(self, f: callable, *variables):
        """
        Initialize reverse class. For detailed implementation see
:py:meth:`AutoDiff.reverse.Reverse.grad`.

        Args:
            f: Function that is callable

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

    @staticmethod
    def grad(f: callable, *variables):
        r"""
        Evaluate the full Jacobian in reverse mode. This is the
method that is used internally
        by :py:meth:`AutoDiff.reverse.Reverse.__init__`.
        For each scalar function, use a forward pass and a reverse
pass.
        Stack the partial derivative columns into the full Jacobian.
```

```
        :param f: A callable function to perform differentaition on
        :type f: function object

        :param variables: The input for variables of function ``f``
        :type variables: integer or float or numpy array or list of
integers or floats

        :return: Jacobian
            Stack the partial derivative columns into the full
Jacobian.
        :rtype: integer or float or numpy array
        """
        # interate through variables and stack the partial derivative
columns into Jacobian
        for var in variables:
            output_depend = []
            var_der = var.grad_vec(output_depend)
            # var_der: ordered list of derivatives of each parent of
var
            # output_depend: ordered list of output rnode that each
parent of var points to
            var.clear() # clear the paths to prepare for the next
iteration of variables
            # sum up the partial derivaties of each scalar function
in var_der
            var_der = df/dvar # details will not be shown here
            ders.append(var_der)

        ders = np.stack(ders, axis = -1) # stack derivatives of each
var
        if isinstance(output, list): # vector function
            values = np.array([o.val for o in output])
            if num_variables == 1:
                ders = ders.T # reshape column vector to 1d array
        else: # scalar function
            values = output.val
            ders = ders.T
            if len(ders) == 1:
                ders = ders[0]
        return values, ders
```

6.2 Optimization

We provide optimizers including Netwon's method and Stochastic Gradient Descent.

- Newton's Method iteratively updates x using the following equation:

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

- Stochastic Gradient Descent (SGD) iteratively updates x using the following equation:

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

6.2.1 Newton's Method

```
import numpy as np
from .. import Forward
def Newton(f: callable, *x0, tol=1e-5, max_iter=1000, n_iter=1):
    r"""
    Newton's method

    To find :math:`\mathbf{x}` such that :math:`F(\mathbf{x}) =
\mathbf{0}`, we need update :math:`\Delta \mathbf{x}_{k}` such that

    .. math::
                    J_F(\mathbf{x}_k)\Delta \mathbf{x}_{k} &= -
F(\mathbf{x}_{k}) \\
                \mathbf{x}_{k+1} &\gets \mathbf{x}_{k} + \Delta
\mathbf{x}_{k}
    where :math:`J_F(\mathbf{x}_k)` is the Jacobian and :math:`\Delta
\mathbf{x}_{k}` is the update.

    :param f: callable, the :math:`F: \mathbb{R}^m \mapsto
\mathbb{R}^n` function
    :param x0: initial guess, note that Newton is quadratic
convergence if initial guess is close to the actual solution
    :param tol: tolerance, the algorithm terminates when it hits the
tolerance i.e. when :math:`\|F(\mathbf{x})\|_F < \text{tol}` is
reached, the algorithm terminates
    :return: the solution :math:`\mathbf{x}`
    """
    n_iter += 1
    if n_iter == max_iter:
        raise RuntimeError(f'The function does not converge in
{n_iter} iterations!')
    x0 = np.array(x0)
    # if the norm of the function is less than the tolerance,
consider the method converged
    if np.linalg.norm(f(*x0)) < tol:
        # if result list has length 1, return the number without the
bracket
        if len(x0) == 1:
            return x0.item()
        return x0
    # use Forward AD for derivative calculation
    g = Forward(f, *x0)
    if isinstance(g.der, (int, float)):
        # if the derivative g is a number, perform Newton's Method in
1D
        new_x = x0 - g.val / g.der
    else:
        # else perform Newton's Method in nD
        if g.der.ndim == 1:
            g.der = g.der.reshape(1, -1)
            g.val = np.array(g.val).reshape(1)
        update, *_ = np.linalg.lstsq(g.der, -g.val, rcond=None)
```

```
        new_x = x0 + update
    return Newton(f, *new_x, tol=tol, n_iter=n_iter)
```

6.2.2 Stochastic Gradient Descent

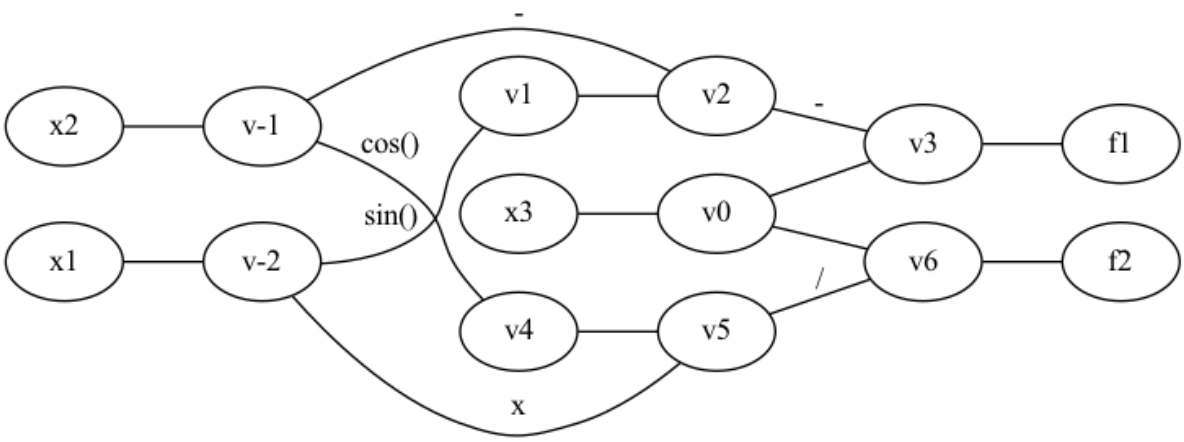
```
import numpy as np
from .. import Forward
def SGD(f: callable, *x0, eta=1e-1, n_iter=50000, tol=1e-5):
    r"""
    Stochastic gradient descent

    It optimizes the following procedure iteratively

    .. math::
        \mathbf{x} \gets \mathbf{x} - \eta \nabla f(\mathbf{x})

    where :math:`f: \mathbb{R}^n \mapsto \mathbb{R}`

    :param f: callable, the :math:`F: \mathbb{R}^n \mapsto
\mathbb{R}` function
    :param x0: initial guess
    :param eta: learning rate :math:`\eta`, which needed to be picked
for each specific optimization task
    :param n_iter: after :code:`n_iter` steps the algorithm will
terminate
    :param tol: the algorithm terminates when it reaches the
tolerance, i.e. when :math:`|f(\mathbf{x})| < \text{tol}` is reached
    :raises RuntimeError: The function does not converge in n_iter
iterations!
    :return: the final solution
    """
    i = 0
    while i < n_iter:
        # if the norm of the function is less than tolerance,
consider the method converged
        if np.linalg.norm(f(*x0)) < tol:
            # if result list has length 1, return the number without
the bracket
            if len(x0) == 1:
                return x0.item()
            return x0
        i += 1
        # use Forward AD for derivative calculation
        g = Forward(f, *x0)
        # apply stochastic gradient descent
        x0 -= eta * g.der
    # if the function does not converge in 50000 iterations, we
consider the function does not converge, can raise a Runtime error
    raise RuntimeError(f'The function does not converge in {n_iter}
iterations!')
```

6.3 Computational Graph Visualization

The `Graphviz` software is used to support the implementation of computation graph visualization. The computation graph is developed by taking the output from the forward mode and recursively iterating through each node's parent and operation until there is no more parent. As we traverse through the parents and operations, we add nodes and edges to the graph. Each node represents a variable or an intermediate result. Each edge specifies the from node and the to node along with the operation that the edge takes on. Two functions are used to implement the visualization functionality: `build_graph(g, output)` and `generate_graph(x, g)`. `build_graph(g, output)` is the helper function that adds all nodes and edges into a graphviz Graph object. `generate_graph(x, g)` is the main visualization function that ties everything together to produce a visualization of the computation graph and then outputs the final computation graph as a png file. For example, given the input function

$$f(x_1, x_2, x_3) = \begin{bmatrix} sin(x_1) - x_2 - x_3 \\ cos(x_2) \times x_1/x_3 \end{bmatrix}$$

After completing the forward mode, `generate_graph(x, g)` can be called to produce the following computation graph visualization.



7.0 Broader Impact and Inclusivity Statement

The potential broader impacts and implications of `AutoDiff` could be significant, as it has the potential to make it easier for researchers and developers to implement automatic differentiation in their own projects. Automatic differentiation is a powerful tool for optimizing machine learning algorithms, which can have a wide range of applications in various fields.

One potential way that people could use `AutoDiff` responsibly is by using it to improve the performance of machine learning algorithms in a way that is transparent and explainable. For example, if `AutoDiff` is used to optimize a predictive model in healthcare, finance, or environmental science, the results of the optimization should be clearly communicated and explained to clinicians and patients or other relevant

stakeholders, so that they can understand the model's predictions and make informed decisions.

On the other hand, there could be potential ethical implications if `AutoDiff` is used irresponsibly or without proper oversight. For example, if a predictive model optimized with `AutoDiff` is used to make decisions that have a significant impact on people's lives, such as in hiring or loan applications, it is important that the model is fair and unbiased. If the model is not properly validated or checked for bias, it could have negative consequences for the individuals who are affected by its decisions.

Overall, it is important for users of AutoDiff to use the software responsibly and take into account the potential broader impacts and ethical implications of their work. By doing so, they can help to ensure that the technology is used for the benefit of society, rather than causing harm.

In terms of inclusivity, one potential way that `AutoDiff` could be inclusive to the broader community is by making automatic differentiation more accessible to a wider range of users. Currently, automatic differentiation can be difficult to implement, especially for users who do not have a strong background in mathematics or computer science. By providing an easy-to-use software package, `AutoDiff` could make automatic differentiation more accessible to a broader range of users, including those from underrepresented groups. This could help to promote diversity and inclusion in the fields of machine learning and artificial intelligence.

Additionally, `AutoDiff` could be inclusive by providing documentation and user support in multiple languages, which would make it more accessible to users who speak languages other than English. In the future, we plan to support more languages such as Chinese and Spanish. This could be especially useful for users in non-English speaking countries, where access to resources and support for machine learning and artificial intelligence can be more limited. By providing support in multiple languages, `AutoDiff` could help to break down language barriers and make it easier for users from diverse backgrounds to access and use the software.

For any developers who would like to build on top of the `AutoDiff` package, they can fork the `AutoDiff` GitHub repository. If they find an issue with the implementation of the `AutoDiff` package or have any suggestions for improvements, they can send a pull request from the forked GitHub respository. All developers of the `AutoDiff` package (i.e., our AC207 project team) will review the pull request and decide whether or not to integrate the code change into the `AutoDiff` package source code. Once all team members review the pull request and confirm that there are no more issues, one of the team members will approve the pull request and merge the pull request into the `main` branch for `AutoDiff`.

8.0 Future Work

Automatic Differentiation can be useful in many fields.

In physics, this can be useful for a number of applications. For example, it can be used to quickly and accurately compute the derivatives of functions that describe physical phenomena, such as the motion of a particle or the evolution of a physical system over time. This can allow physicists to more easily solve complex problems, and to make more precise predictions about the behavior of physical systems. Additionally, auto differentiation can be used to optimize physical models, by adjusting model parameters in such a way as to best fit experimental data.

In the process of gene expression, where genes are transcribed into RNA and then translated into proteins, can be modeled using differential equations. By using automatic differentiation, researchers can quickly and accurately compute the derivatives of these equations, which can provide important insights into the underlying mechanisms of gene expression. Additionally, automatic differentiation can be used to train machine learning models that can be used for tasks such as predicting the effects of genetic mutations or identifying patterns in large genomic datasets.

Automatic Differentiation can also be widely used in applied mathematics field. For example, in fluid dynamics, the gradient of the water flow needs to be calculated in order to analyze velocities and pressures in a 3D space. The application of fluid dynamics is widely spreaded from analyzing dynamics for aircrafts to determing the flow rate through water pipelines.

In the field of public health, automatic differentiation can be used in the study of the diffusion model. For example, we can trace the spreading speed of the virus. This would be helpful for the scientists to generate valid and practical advice for the government in the process of establishing public health policies.

While many exciting projects can be built on top of our `AutoDiff` package, there are still many additional features that we can improve in the package itself.

The first limitation is that the current Reverse Mode implementation for automatic differentiation does not support functions with nested variable dependency. Currently, users have to define the function with no intermediate lines. One future improvement is to enable support for nested dependency so that there is a fully functional Reverse Mode that supports any functions.

Secondly, a full-fledged backpropagation support could be provided as an additional feature. Backpropagation used in deep learning generally involves matrix or tensor. However, the current `AutoDiff` implementation does not support those kinds of input. In the future we will extend our package to support those data inputs and provide a convenient wrapper for commonly-used operations such as linear map, convolution layers and attention module, as well as activation function and several loss functions.

Additionally, two optimizers are provided as additional features, namely Newton method and SGD. However there are many more interesting optimizers such as Adam and AdaFactor. Additional optimizers can also be implemented for the `AutoDiff` package as a future step.

Moreover, the current computation graph visualization is static, meaning that the generated computation graph is a fixed image. This visualization can be made more interactive, for example, users could drag the nodes or modify the graph on the fly.

Lastly, since the `AutoDiff` package supports computing gradient and Jacobian, there are many interesting gradient based samplings to explore. For example, Langvian dynamics samples from gradients of evidence. Sampled Langvian dynamics is a path towards the mode of two Gaussian mixtures. Providing those sampling to users could also be a nice functionality to have.

9.0 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.