UNIVERSITY OF CAMBRIDGE

Automatic Differentiation With the dual_autodiff Package

C1 Research Computing Coursework Report

PRITHVI SINGH
CRSID: ps2012

word count: 2753 words

Michaelmas term 2024-25

Contents

1 Introduction

The back propagation algorithm is the backbone of modern Artificial intelligence. It is in essence the chain rule on steroids, but when we have to calculate several complex derivatives we run into a crucial complication, the problem becomes too complex for a symbolic approach and a numerical method introduces far too much error, our savior in these circumstances is automatic differentiation, an ingenious method to calculate derivatives with machine precision, but without the headache of having to do a bunch of analytical calculation for the same. In this project we have tried to make a package that performs automatic differentiation using dual numbers. This is called forward mode automatic differentiation and is a crucial piece of the puzzle for implementing AI networks at the large scale.

2 The Dual class

Dual numbers are a mathematical construct similar to complex numbers but with key differences in their structure and application. A dual number is expressed as

$$x = a + b\epsilon,$$

where a is the real part, b is the dual part, and ϵ is a special unit with the property $\epsilon^2 = 0$. This unique property makes dual numbers particularly powerful for performing automatic differentiation (AD).

Automatic differentiation is a method for computing derivatives of functions accurately and efficiently. Using dual numbers, derivatives are computed as a byproduct of evaluating a function. For a function $f(x)$, extending it to operate on dual numbers results in

$$f(a + b\epsilon) = f(a) + f'(a)b\epsilon,$$

where $f(a)$ is the function value, and $f'(a)$ is its derivative. By setting $b = 1$, the dual part directly represents the derivative at $x = a$.

This approach is especially valuable in computational science, as it avoids the inaccuracies of numerical differentiation and is computationally cheaper than symbolic differentiation. Forward-mode AD, based on dual numbers, is widely used in machine learning, optimization, and scientific computing, enabling efficient gradient computations critical for training deep neural networks and solving complex mathematical problems.

In this project we make a Python class called the dual class. It initiates our dual number objects and performs computation on them. attached is basic code for the dual class and some of its primitive methods.

```python
import numpy as np

class Dual:
    '''
    Python class to perform automatic differentiation using dual numbers.
    arguments:
    a: input value
    b: input slope
    '''
```

```
# Initialisation
def __init__(self, a, b):
    '''
    Initializes a Dual number with a real part and a dual part.

    Parameters:
    a (float): The real part of the dual number.
    b (float): The dual part (slope) of the dual number.
    '''
    self.real = a
    self.dual = b

# Defining what to return when a dual object is printed
def __repr__(self):
    '''
    Returns a string representation of the Dual number for debugging.

    Returns:
    str: A string in the format (real, dual).
    '''
    return f"({self.real},{self.dual})"

def __str__(self):
    '''
    Returns a user-friendly string representation of the Dual number.

    Returns:
    str: A string in the format Dual(real=real_value, dual=dual_value).
    '''
    return f"Dual(real={self.real},dual={self.dual})"

def __class__(self):
    '''
    Returns the class name of the Dual object.

    Returns:
    str: The name of the class.
    '''
    return "Dual"

# Defining arithmetic operations
def __add__(self, o):
    '''
    Defines addition for Dual numbers.

    Parameters:
    o (Dual): Another Dual number to add.

    Returns:
    Dual: A new Dual number representing the sum.
    '''
```

```
        return Dual(self.real + o.real, self.dual + o.dual)

    def __sub__(self, o):
        '''
        Defines subtraction for Dual numbers.

        Parameters:
        o (Dual): Another Dual number to subtract.

        Returns:
        Dual: A new Dual number representing the difference.
        '''
        return Dual(self.real - o.real, self.dual - o.dual)

    def __mul__(self, o):
        '''
        Defines multiplication for Dual numbers.

        Parameters:
        o (Dual): Another Dual number to multiply.

        Returns:
        Dual: A new Dual number representing the product.
        '''
        return Dual(self.real * o.real, self.dual * o.real + self.real * o.dual)

    def __truediv__(self, o):
        '''
        Defines division for Dual numbers.

        Parameters:
        o (Dual): Another Dual number to divide by.

        Returns:
        Dual: A new Dual number representing the quotient.
        '''
        return Dual(self.real / o.real, (self.dual * o.real - self.real * o.dual) / (o.real
```

Once we have created our class we then do some mathematical operations to make sure that our logic is sound:

```
x=Dual(2.,1.)
print(x)
```

Yields:

```
Dual(real=2.0,dual=1.0)
```

```
x=Dual(2.,1.)
print(x)
```

Yields:

```
Dual(real=5.0,dual=3.0)
```

```
print(x.sin())
```

Yields:

```
Dual(real=0.9092974268256817,dual=-0.4161468365471424)
```

This can be confirmed in the complementary Jupyter Notebook. later we will implement an entire test suite to make sure our class behaves as it should, but for now we will move on to implementing this class as a package.

3 Creating a Package

Once we have populated our class with the methods we need we will move on to making a python package for it.

We do this because a package facilitates code organization, reusability, and distribution. A package typically includes the following components:

- Modules: These are .py files containing Python code. Each module performs specific tasks, such as defining classes, functions, or constants.
- __init__.py: This file marks a directory as a Python package. It can be empty or contain initialization code for the package, such as imports or setup logic.
- Project Configuration: The pyproject.toml file is the modern configuration standard for defining the package build system, metadata, and dependencies. Alternatively, older setups use setup.py for package configuration and setup.cfg for declarative metadata.
- Tests: A tests/ directory contains test modules to validate the functionality of the package. Tools like pytest or unittest are often used.
- Documentation: This includes README files (e.g., README.md or README.rst) and detailed documentation often generated using tools like Sphinx.
- We also include other optional files that are nice to have: requirements.txt: Specifies package dependencies. LICENSE: States the legal usage rights.

Organized correctly, these parts make a Python package robust, maintainable, and easy to distribute via tools like pip or platforms like PyPI. Our package today will support Python but also Cython.

To make our package we will write our dual class into a dual.py file, the convention is that filenames begin with small lectures but class names can be capitalized when convenient.

The pyproject.toml file is a key configuration file for Python projects, introduced as part of PEP 518. It defines metadata, dependencies, and build system requirements for the project. Below is an overview of what we include in a pyproject.toml file and their purposes:

- Build System Requirements This section specifies the build backend and the tools required to build the project.

```
[build-system]
requires = ["setuptools","wheel","setuptools_scm"]
build-backend = "setuptools.build_meta"
```

- Project Metadata This section defines the project's metadata and is used by tools like PyPI.

```
[project]
name = "dual_autodiff"
authors = [
    { name = "Prithvi Singh", email= "prithvisindhu9@gmail.com" },
]
description = "Perform forward mode automatic differentiation using dual numbers"
readme = "README.md"
version= "0.1.0"
license= {file="LICENSE.txt"}
classifiers = [
    "Development Status :: 2 - Pre-Alpha",
    "Operating System :: OS Independent",
    "Environment :: Console",
    "Intended Audience :: Data Science/Research",
    "Topic :: Scientific/Engineering :: Machine Learning",
    "Programming Language :: Python :: 3.8",
    "Programming Language :: Python :: 3.9",
    "Programming Language :: Python :: 3.10",
    "Programming Language :: Python :: 3.11",
    "Programming Language :: Python :: 3.12",
    "Programming Language :: Python :: 3.13"
]
```

- Tool-Specific Configurations, and dependency information

```
]
requires-python = ">=3.8.0"
dependencies = [
    "numpy>=1.23.0"
]
[project.optional-dependencies]
GUI = ["PySide6>=6.1"]
docs = ["sphinx", "sphinx_rtd_theme>=1.0", "sphinxcontrib-jquery"]

[tool.setuptools.packages.find]
where = ["."]

[tool.pytest]
addopts = "-v"
testpaths = ["/tests"]
```

4 Installation and testing

Once we created the structure for our package we can install it using the python command:

```
pip install -e .
```

The '-e' makes sure that our package is installed in editable mode, meaning that any changes that we make in it source will reflect in our program on the fly.

Great! after installing out package we can use it to differentiate a function. We will compare three methods, analytical derivatives calculated by hand, a numerical finite difference method and our automatic differentiation package.

```
#Importing other necessary packages
from dual_autodiff.dual import Dual
import numpy as np
import matplotlib.pyplot as plt

# Analytic function
def f(x):
    return np.log(np.sin(x)) + x**2*np.cos(x)
# Analytic derivative
def f_(x):
    return 1/np.tan(x) + 2*x*np.cos(x) - x**2*np.sin(x)
# Numeric derivative
def numeric_diff(f,x,h=1e-6):
    return (f(x+h)-f(x-h))/(2*h)

x= Dual(1.5,1)

diff= x.sin().log() + x**2*x.cos()
print(diff)

print(f(1.5),numeric_diff(f,1.5,1e-7))

print(f(1.5),f_(1.5))
```

```
Dual(real=0.15665054756073515,dual=-1.9612372705533612)
0.15665054756073515 -1.9612372717525695
0.15665054756073515 -1.9612372705533612
```

It is reassuring to see that we get the same result, however, when we time the three functions we find that our automatic differentiation is much slower than an analytic approach and even a little behind our numeric approach, so what is the benefit then? The benefit is that our dual class is much more precise than the numeric approach. Let us see how error scales with the step size in numeric differentiation:

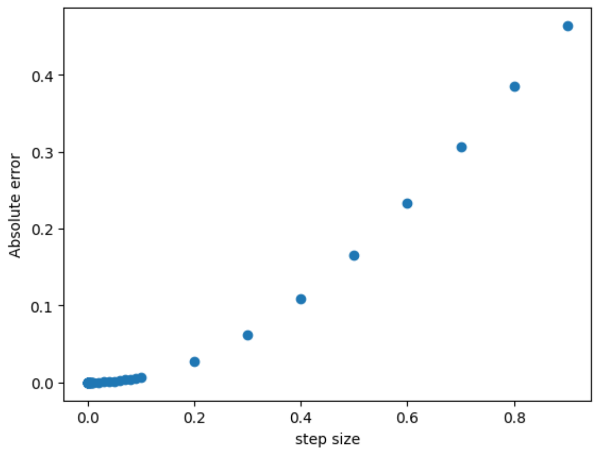

Figure 1: absolute error with respect to step size

The plot above showcases the Achilles heal of numeric differentiation, it is not as precise as an analytic method, in fact even for a second order finite difference scheme like ours the error scales the with square of the step size and accumulates with subsequent calculations. When we need to find the minima of very complex loss landscapes this property makes numerical derivatives extremely inaccurate, and practically unusable. Here lies the benefit of dual numbers, they are not exactly numeric methods, they are just a clever utilization of the chain rule that allow us to get results with machine precision for any complex landscape as long as its symbolic building blocks are defined by us.

4.1 Implementing a Test Suite

Now that we have built our package we want to make sure that it performs as expected, to confirm that we do not have any buggy code or something of the sort the leads to an error. To do this we implement a test suit that we will run with the help of pytest. Some examples in our test suit are as follows:

```
import numpy as np
from dual_autodiff.dual import Dual

class TestDual:
    def test_initialization(self):
        d = Dual(3.0, 2.0)
        assert d.real == 3.0
        assert d.dual == 2.0

    def test_repr(self):
        d = Dual(3.0, 2.0)
        assert repr(d) == "(3.0,2.0)"

    def test_str(self):
        d = Dual(3.0, 2.0)
        assert str(d) == "Dual(real=3.0,dual=2.0)"

    def test_addition(self):
        d1 = Dual(3.0, 2.0)
        d2 = Dual(1.0, 4.0)
        result = d1 + d2
        assert result.real == 4.0
        assert result.dual == 6.0

    def test_subtraction(self):
        d1 = Dual(3.0, 2.0)
        d2 = Dual(1.0, 4.0)
        result = d1 - d2
        assert result.real == 2.0
        assert result.dual == -2.0

    def test_multiplication(self):
        d1 = Dual(3.0, 2.0)
        d2 = Dual(1.0, 4.0)
        result = d1 * d2
        assert result.real == 3.0
        assert result.dual == 14.0  # 2*1 + 3*4

    def test_division(self):
        d1 = Dual(3.0, 2.0)
        d2 = Dual(1.0, 4.0)
        result = d1 / d2
        assert result.real == 3.0
        assert result.dual == -10.0  # (2*1 - 3*4) / (1^2)
```

Each test is meant to ensure that a particular method in the class is running as expected. If every method is correct, we can rest assured that our dual numbers will be precise on any function, as every function that it can solve is built by piecing together our basic functions, which we have defined methods for.
To test our functions we will employ Pytest, which is a powerful and flexible testing framework that simplifies writing and running test cases. It automatically discovers test files and functions following the naming convention test_*.py and test_*.py. As long as the tests are written as assertion functions pytest can run them one by one and check if they pass or fail. To run pytest on our suite we call it from the command line with the phrase "pytest -s test."

5 Documentation

For documentation we use the sphinx package, which is ideal since it converts reStructuredText (.rst) or Markdown (.md) files into HTML. Sphinx is particularly effective for Python packages as it can automatically extract information from doc-strings in our code, making it easier to document our API and libraries.
To use Sphinx, we initialize a Sphinx project in our package directory by running sphinx-quickstart, which sets up the necessary files and directories (conf.py and index.rst). Our conf.py file controls how Sphinx generates our documentation. It defines settings, extensions, and paths that customize the documentation output. The index.rst file acts as the root of our documentation. It's the first page that Sphinx processes when generat-

ing the documentation and usually contains the general structure of the documentation, including the table of contents, intro text, and links to other sections of the docs. This documentation can then be hosted online by a range of methods, of which we will choose the readthedocs website. We simply add a readthedocs.yaml file where we configure how readthedocs should treat our sphinx data and the rest is quite seamless. For the purpose of this project however we can simply view the documentation as a html file, which is stored within the '_build' folder after we use the command 'make html'.

6 Cythonizing our Package

To Cythanize our package we must follow the following steps:

- **Step 1: Install Cython**
  - Open terminal or command prompt
  - Run the following command to install Cython:

```
pip install cython
```

- **Step 2: Create a `setup.py` File**
  - Create a `setup.py` file that tells setuptools how to build the Cython extension.
  - Example `setup.py`:

```
from setuptools import setup, Extension
from Cython.Build import cythonize

ext_modules = [
    Extension(
        name="dual_x",
        sources=["dual_autodiff_x/dual.pyx"],
    ),
]

setup(
    ext_modules=cythonize(ext_modules),
    zip_safe=False,
    package_dir={"": "dual_autodiff_x"},
)
```

- **Step 3: Build the Cython Extension**
  - Open terminal or command prompt
  - Navigate to the project directory (where `setup.py` is located)
  - Run the following command:

```
python setup.py build_ext --inplace
```

- **Step 4: Check the Output**
  - Verify that the compiled `.so` or `.pyd` file exists in the `dual_autodiff_x/` folder.
  - Example directory structure:

```
dual_autodiff_x/
 dual_autodiff_x/
    __init__.py
    dual.pyx
    dual.cpython-39-x86_64-linux-gnu.so   # Compiled file
 setup.py
```

After cythonizing our package we can test its performance compared to pure Python. The results of such a test can be seen in the documentation of this package where the cythonized package has been compared to the pure python implementation. Here are the results of the difference in execution times for 100,000 runs of each of these calls:

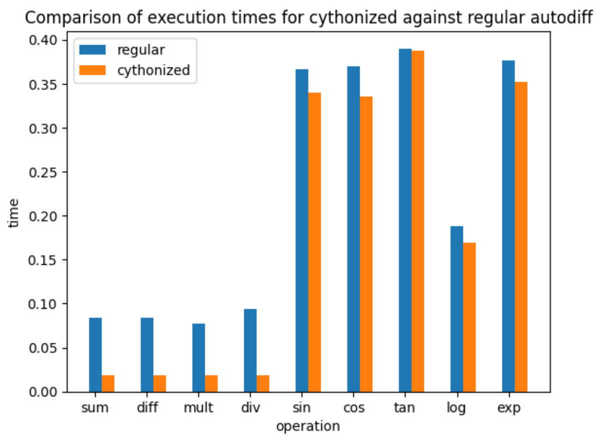

Figure 2: Comparing Cythonized performance against pure Python

It can be inferred from this graph that cythonizing does not improve performance across the board, as our methods for algebraic expressions like sin, cos and log are written in numpy which is, to begin with, a C wrapper. Therefore, it stands to reason that we do not see much of a performance boost in these cases. However, we do see a significant performance boost (up to 4x) when comparing arithmetic operations, as the implementation of those is using pure python and the static types in C help speed these up significantly.

7 Making wheels

To make a wheel from our package we use Docker, which helps set up a virtual environment on our machine. We follow the following methodology:

- **Step 1: Create a `Dockerfile`**
  - In our project directory, we create a `Dockerfile` that defines the steps for building the wheel inside a Docker container.

- `Dockerfile`:

```
# Use official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any necessary dependencies
RUN pip install --upgrade pip
RUN pip install -r requirements.txt

# Build the wheel
RUN python setup.py bdist_wheel
```

- **Step 2: Build the Docker Image**
  - Open terminal or command prompt
  - Navigate to our project directory (where the `Dockerfile` is located)
  - Build the Docker image by running the following command:

    ```
    docker build -t dual_autodiff:latest .
    ```

  - This will create a Docker image tagged `dual_autodiff:latest`.
- **Step 3: Run the Docker Container**
  - Run the Docker container to build the wheel:

    ```
    docker run --rm -v $(pwd)/dist:/app/dist dual_autodiff:latest
    ```

  - This will mount our local `dist/` directory into the container, where the wheel will be stored after building.
- **Step 4: Check the Output**
  - After running the container, check the `dist/` directory in our local project directory.
  - we should see the generated wheel file (e.g., `dual_autodiff-0.1-py3-none-any.whl`).

We follow these steps to build our wheel which can then be distributed through github or gitlab.

7.1 Building Wheels for Cythonized Packages

The process for building a wheel for a cython package is slightly different as the distribution of a cython package differs based on the operating system, we can distributed the our entire package as is but any user would then need a C compiler to build the cythonized package for their device. For example, cythonized projects on Windows are run through a .pyd file, while on Linux they require a .so file, this difference in architecture means that if we want to make a wheel for our package, it has to be device dependent.

To build our wheel we will use cibuildwheel which helps us build wheels for precisely the distribution we want. For this project we will use the following commands commands to build wheels for our required distributions:

```
CIBW_BUILD="cp310-manylinux_x86_64" CIBW_ARCHS="x86_64" cibuildwheel
--platform linux

CIBW_BUILD="cp311-manylinux_x86_64" CIBW_ARCHS="x86_64" cibuildwheel
--platform linux
```

These wheels are uploaded along with this report and can be installed to test the notebook.

8 Conclusion

In this project we built a package to compute automatic differentiation using dual numbers. We made a class that does our computation and then gave our project structure and documentation. We then cythonized the project, which led to a significant speed-up for pure python implementations, and then built wheels that can be distributed. In this way we covered the entire methodology from start to finish of building a package in Python.

9 Appendix

9.1 Use of AI tools

AI has been incorporated to write the documentation for this project, especially for the task of Cythonizing and building wheels. However, its implementation has been checked and absorbed and it has been used as a tool for learning.