CS207 - Milestone1

Chen Shi, Stephen Slater, Yue Sun

October 2018

1 Introduction

Differentiation, i.e. finding derivatives, has long been one of the key operations in computation related to modern science and engineering. In optimization and numerical differential equations, finding the extrema will require differentiation. There are many important applications of automatic differentiation in optimization, machine learning, and numerical methods (e.g., time integration, root-finding). This software library will use the concept of automatic differentiation to solve differentiation problems in scientific computing.

2 Background

The chain rule, gradient (Jacobian), computational graph, elementary functions and several numerical methods serve as the mathematical cornerstone for this software. The mathematical concepts here come from CS 207 Lectures 9 and 10 on Autodifferentiation.

2.1 The Chain Rule

The chain rule is critical to AD, since the derivative of the function with respect to the input is dependent upon the derivative of each trace in the evaluation with respect to the input.

If we have $h(u(x))$ then the derivative of h with respect to x is:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u} \cdot \frac{\partial u}{\partial x} \tag{1}$$

If we have another argument $h(u, v)$ where u and v are both functions of x, then the derivative of $h(x)$ with respect to x is:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u} \cdot \frac{\partial u}{\partial x} + \frac{\partial h}{\partial v} \cdot \frac{\partial v}{\partial x} \tag{2}$$

2.2 Gradient and Jacobian

If we have $x \in \mathbb{R}^m$ and function $h\left(u\left(x\right), v\left(x\right)\right)$, we want to calculate the gradient of h with respect to x:

$$\nabla_x h = \frac{\partial h}{\partial u} \nabla_x u + \frac{\partial h}{\partial v} \nabla_x v \tag{3}$$

In the case where we have a function $h(x) : \mathbb{R}^m \to \mathbb{R}^n$, we write the Jacobian matrix as follows, allowing us to store the gradient of each output with respect to each input.

$$\mathbf{J} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_m} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial x_1} & \frac{\partial h_n}{\partial x_2} & \cdots & \frac{\partial h_n}{\partial x_m} \end{bmatrix}$$

In general, if we have a function $g(y(x))$ where $y \in \mathbb{R}^n$ and $x \in \mathbb{R}^m$. Then g is a function of possibly n other functions, each of which can be a function of m variables. The gradient of g is now given by

$$\nabla_x g = \sum_{i=1}^{n} \frac{\partial g}{\partial y_i} \nabla_x y_i(x). \tag{4}$$

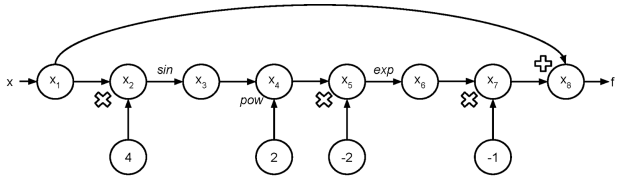
2.3 The Computational Graph

The computational graph lets us visualize what happens during the evaluation trace. The following example is based on Lectures 9 and 10. Consider the function:

$$f(x) = x - \exp\left(-2\sin^2(4x)\right)$$

If we want to evaluate f at the point x, we construct a graph where the input value is x and the output is y. Each input variable is a node, and each subsequent operation of the execution trace applies an operation to one or more previous nodes (and creates a node for constants when applicable).

Figure 1: Sample computational graph for $f(x) = x - \exp\left(-2\sin^2(4x)\right)$.



As we execute $f(x)$ in the "forward mode", we can propagate not only the sequential evaluations of operations in the graph given previous nodes, but also the derivatives using the chain rule.

2.4 Elementary functions

An elementary function is built up of a finite combination of constant functions, field operations $(+, -, \times, \div)$, algebraic, exponential, trigonometric, hyperbolic and logarithmic functions and their inverses under repeated compositions. Below is a table of some elementary functions and examples that we will include in our implementation.

Elementary Functions	Example
powers	x^2
roots	$\sqrt{x}$
exponentials	e^x
logarithms	$\log(x)$
trigonometrics	$\sin(x)$
inverse trigonometrics	$\arcsin(x)$
hyperbolics	$\sinh(x)$

3 How to Use *DeriveAlive*

The user will use `pip` to install the package. After installation, the user will need to import our package (see psuedocode below). Implicitly, this will import other dependencies (such as `numpy`), since we include those dependencies as imports in our module. Then, the user will define an input of type `Var` in our module. After this, the user can define a function in terms of this `Var`. Essentially, the user will give the initial input

x and then apply f to x and store the new value and derivative with respect to x inside f. At each step of the evaluation, the program will process nodes in the implicit computation graph in order, propagating values and derivatives. The final output yield another `Var` containing $f(x)$ and $f'(x)$.

For example, consider the case $f(x) = \sin(x)+5$. If the user wants to evaluate $f(x)$ at $x = a$, where $a = \pi/2$, the user will instantiate a `Var` object as `da.Var(np.pi/2)`. Then, the user will give the initial input a and set $f1 = f(a)$, which stores $f(a)$ and $f'(a)$ as attributes inside the `Var` $f1$. This functionality will propagate throughout the graph with more variables in a recursive structure, where each evaluation trace creates a new `Var`. See the code below for a demonstration. Note that the $\sin(\pi/2) = 1.0$, and the derivative is 0. Also note that we can assign $f1 = f(a)$ by explicitly applying the operations of f to a without creating an intermediary f.

```
## install at command line
pip install DeriveAlive

## Python
Python
>>> import DeriveAlive as da
>>> import numpy as np

# Expect value of 18.42, derivative of 6.0
>>> x1 = da.Var(np.pi / 2)
>>> f1 = 3 * 2 * x1.sin() + 2 * x1 + 4 * x1 + 3
>>> print (f1.val, f1.der)
18.42477796076938 6.0

# Expect value of 0.5, derivative of -0.25
>>> x2 = da.Var(2.0)
>>> f2 = 1 / x2
>>> print (f2.val, f2.der)
0.5 -0.25

# Expect value of 1.5, derivative of 0.5
>>> x3 = da.Var(3.0)
>>> f3 = x3 / 2
>>> print (f3.val, f3.der)
1.5 0.5
```

4 Software Organization

4.1 Current Directory Structure

```
cs207-FinalProject/
                    README.md
                    LICENSE
                    DeriveAlive.py
                    docs/
                         milestone1.pdf
                    tests/
                          test_DeriveAlive.py
                    ...
```

4.2 Modules plan on including and their basic functionality

- `NumPy` - This provides an API for a large collection of high-level mathematical operations. In addition, it provides support for large, multi-dimensional arrays and matrices.
- `doctest` - This module searches for pieces of text that look like interactive Python sessions (typically within the documentation of a function), and then executes those sessions to verify that they work exactly as shown.
- `pytest` - This is an alternative, more Pythonic way of writing tests, making it easy to write small tests, yet scales to support complex functional testing. We plan to use this for a comprehensive test suite.
- `setuptools` - This package allows us to create a package out of our project for easy distribution. See more information on packaging instructions here: `https://packaging.python.org/tutorials/packaging-projects/`.

4.3 Where will your test suite live?

Our test suite will be in a test file in its own `tests` folder. We will use Travis CI for automatic testing for each push, and Coveralls for line coverage metrics. We have already set up these integrations, with badges included in the `README.md`.

4.4 How will you distribute your package?

We will use Python Package Index (PyPI) for distributing our package. PyPI is the official third-party software repository for Python and primarily hosts Python packages in the form of archives called sdists (source distributions) or precompiled wheels.

5 Implementation

We plan to implement the forward mode of autodifferentiation with the following choices:

- Core data structures: The core data structures will be classes, lists and numpy arrays.
  - Classes will help us provide an API for differentiation and custom functions, including custom methods for our elementary functions.
  - Lists will help us maintain the collection of trace variables and output functions (in the case of multi-output models) from the computation graph in order. For example, if we have a function $f(x) : \mathbb{R}^1 \to \mathbb{R}^2$, then we store $f = [f1, f2]$, where we have defined $f1$ and $f2$ as functions of x, and we simply process the functions in order. We will also use lists as a class attribute within each `Var` to keep track of the derivatives with respect to each input variable, where the length of the list is the number of input variables in the function. Depending on the extensions we choose for the project, we may use lists to store the parents of each node in the graph.
  - Numpy arrays are included for operations on the Jacobian, in case we need to store all possible derivatives in one data structure – this is especially helpful in the multi-input and multi-output function case (vector-vector functions). If we want to optimize our computation, we can store the list of derivatives as a numpy array so that we can apply entire functions to the array, rather than to each entry separately. In the vector-vector case, if we have a function $f : \mathbb{R}^m \to \mathbb{R}^n$, we can process this as $f = [f_1, f_2, \ldots, f_n]$, where each f_i is a function $f_i : \mathbb{R}^m \to \mathbb{R}$. Our implementation can act as a wrapper over these functions, and we can evaluate each f_i independently, so long as we define f_i in terms of the m inputs.
- Our implementation plan currently includes 1 class, but as we progress through the project, we are considering changing this to 2 classes for efficiency purposes:

  - `Var` class. The class instance itself has two main attributes: the value and the evaluated derivatives with respect to each input. Within the class we redefine the elementary functions and basic algebraic functions, including both evaluation and derivation. Since our computation graph includes "trace" variables, this class will account for each variable. Similar to a dual number, this class structure will allow us easy access to necessary attributes of each variable, such as the trace evaluation and the evaluated derivative with respect to each input variable. This trace table would also be of possible help in future project extensions.

  - `Operations` class (maybe). We currently store the operations within each `Var` instance, but in order to avoid duplicating these, we may separate the code to include all operations in their own class.

- Methods and class attributes:

  - We will overload elementary mathematical operations such as addition, division, sine, etc. that will take in 1 `Var` type, or 2 Var types, or 1 Var and 1 constant, and return a new `Var` (i.e. the next "trace" variable). All other operations on constants will use the standard Python library. In each `Var`, we will store attributes of the value of the variable (which is calculated based on the current operation and previous trace variables) and the evaluated gradient of the variable with respect to each input variable.
  - Methods in `Var`: `__add__`, `__radd__`, `__mul__`, `__rmul__`, `__truediv__`, `__rtruediv__`, `__sin__`, `__cos__`, `__tan__`, `__pow__`, `__log__`, `__exp__` (more to be included)
  - Attributes in `Var`: `self.var`, `self.der`. To cover the case of multiple inputs, we plan to implement our `self.der` as a list, in order to account for derivatives with respect to each input variable.

- External dependencies:

  - Modules: `NumPy`, `doctest`, `pytest`, setuptools
  - Test suites: Travis CI, Coveralls

- How will we deal with elementary functions like sin and exp?

  - To handle these functions, we will define our own custom elementary functions within the `Var` class so that we can calculate the $\sin(x)$ of a variable x using a package such as `numpy`, and also store the proper gradient $(-\cos(x)dx)$ to propagate the gradients forward. For example, consider a univariate function where `self.val` contains the current evaluation trace and `self.der` is a numpy array of the derivative of the current trace with respect to the input. When we apply sin, we propagate as follows:

```
def sin(self):
    val = np.sin(self.val)
    der = np.cos(self.val) * self.der
    return Var(val, der)
```

6 Sources

- CS 207 Lectures 9 and 10 (Autodifferentiation)
- Elementary functions: `https://en.wikipedia.org/wiki/Elementary_function`
- Package distribution: `https://packaging.python.org/tutorials/packaging-projects/`