Milestone 1

Introduction

Derivatives play a central role across the sciences, engingeering, technology, and mathematics (STEM) fields. For instance, the calculation of derivatives is necessary for training deep neural networks (i.e., in back propogation), optimizing functions, building economic models, modeling disease spread (e.g., of COVID-19), and examining change in complex systems over time (e.g., physical systems, biological systems). Unfortunately, it is often impossible or infeasible to compute derivatives analytically in real-life problems due to their complexity.

One potential solution to this problem is the finite-difference method, which is a numerical method for approximating the solution to differential equations. The finite-difference method relies on the definition of a derivative to approximate its solution:

$$\frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Of note, the finite-difference method relies on choosing the best ϵ value, and it is unclear how to do so. Poorly-chosen ϵ values may cause inaccurate approximations or instability of solutions (e.g., due to floating point errors). Therefore, another potential method of finding derivatives in real-world applications is symbolic differentiation, which works directly with mathematical equations. However, there are important limitations to this method as well. Specifically, symbolic differentiation can be too computationally costly when functions become highly complex, is a very memory-intensive process, and may not always be applicable depending on the exact form of the function.

These limitations motivate the use of **automatic differentiation (AD)**. AD is a method that is capable of evaluating the derivative of a function specified by a computer program at machine precision, which does not rely entirely on symoblic math (like symbolic differentiation) nor on evaluating the original function at sample points (like the finite-difference method). Rather, AD uses the chain rule to break complex functions into smaller pieces and evaluates only elementary operations at each step. Thus, AD is more easily applicable to a wide range of complex functions suitable for use in real-life applications.

Our package `Pomeranian` will be a PyPI-distributed Python library to perform AD. Just as pomeranians are known for their intelligence, friendliness, and confidence in tackling challenging problems (e.g., taking on big dogs despite their small size), our package will also be user-friendly, powerful, and able to compute solutions to complex problems using simple/elementary operations.

Below, we describe the mathematical background and concepts underlying AD, as well as the usage, software organization, implementation, and licensing of our `Pomeranian` package.

Background

Chain rule

Chain rule is at the heart of automatic differentiation. It enables us to decompose complex functions into piecewise evaluation. Suppose that we have a function $g(h(x))$. We could apply chain rule to calculate the derivative of g with respect to t:

$$\frac{dg}{dx} = \frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$

Chain rule can also be used in a high dimensional scenario. If we have a function $g(h(x))$ where $h \in R^n$ and $x \in R^m$. The derivative can be expressed as a gradient vector:

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

Elementary functions

A complex function could be broken down into elementary functions in order to evaluate them piecewise. Elementary functions is a function of a single variable that is defined as taking sums, products, roots, and other functions.

Forward mode

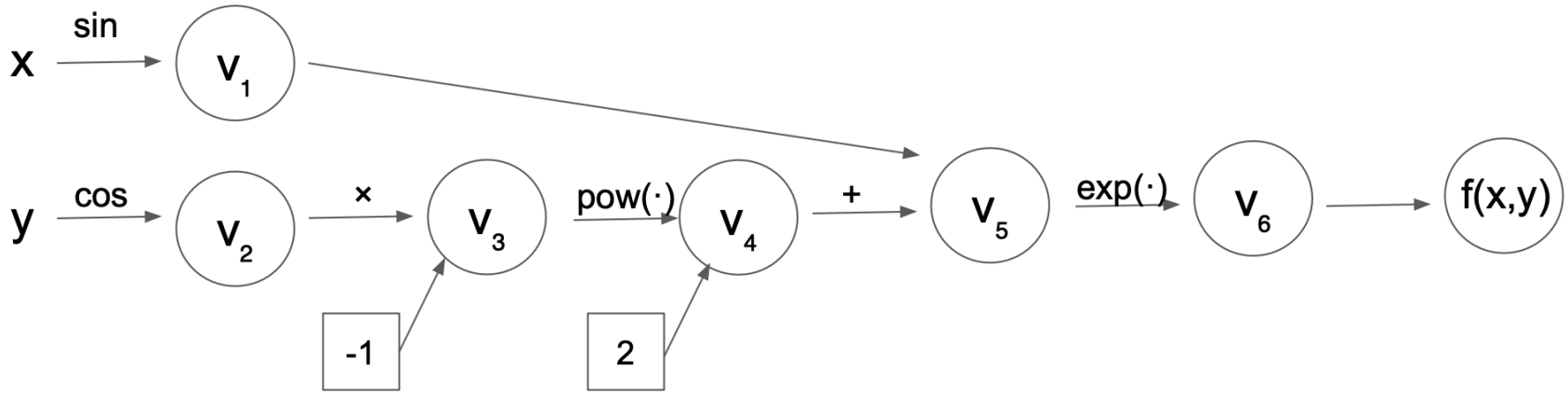
In forward mode, we evaluate the intermediate results v_i and the directional derivative at the same time.

Evaluation trace

The evaluation of a function involves partial ordering of the operations associated with f, forming a evaluation trace. The evaluation trace introduces intermediate results v_i that elementary functions could operate on. These intermediate results depend on independent variables.

Computational graph

We could visualzie the evaluation trace as a computational graph, with each intermediate variable as a node and each elementary function as a edge. For example, the computational graph for $f(x, y) = e^{sin(x) - cos(y)^2}$ looks like the graph below:



Directional derivative and seed vector

In the computational graph for the forward mode, each node not only carries the evaluation of intermediate variable, but also a directional derivative of the intermediate variable in a given direction $p \in R^m$. These two operations happen simultaneously and are termed as the primal trace and the tangent trace. The directional derivative is calculated by projecting the gradient vector into the direction of the seed vector p:

$$D_p g_i \stackrel{\text{def}}{=} \nabla_x g = \sum_{i=1}^{n} \frac{\partial g}{\partial h_i} \nabla h_i(x)$$

In other words, the forward mode AD computes the inner product of the Jacobian with the seed vector p ($J \in R^{n \cdot m}, p \in R^m$)

$$J \cdot p$$

which can be interpreted as projecting Jacobian in the direction given by p. The full Jacobian can be calculated in forward mode AD using m passes, where seed vectors p are set to the m-th unit vector along coordinate x_m for the m-th pass.

The evaluation trace of function $f(x, y) = e^{sin(x) - cos(y)^2}$ at $(x, y) = (\pi/2, \pi/3)$ is as the table follows:

Trace	Elementary Function	Value	Elementary Function Derivative	∇x value	∇y value
x	$\pi/2$	$\pi/2$	1	1	0
y	$\pi/3$	$\pi/3$	1	0	1
v_1	$sin(x)$	0	$cos(x) D_p v_1$	0	0

Trace	Elementary Function	Value	Elementary Function Derivative	∇x value	∇y value
v_2	$cos(y)$	0.5	$-sin(y)D_pv_2$	0	$-\sqrt{3}/2$
v_3	v_2^2	0.25	$2v_2D_pv_3$	0	$-\sqrt{3}$
v_4	$-v_3$	-0.25	$-D_pv_4$	0	$\sqrt{3}$
v_5	v_1+v_4	-0.25	$D_pv_1+D_pv_4$	0	$\sqrt{3}$
v_6	e^{v_5}	$e^{-0.25}$	$e^{v_5}D_pv_5$	0	$\sqrt{3}e^{\sqrt{3}}$

As observed from the table, in the forward mode AD, we are only working with elementary functions whose derivatives are known. therefore, it is trivial to calculate D_pv_j

Reverse mode

The table below shows some major differences between the forward and the reverse mode:

forward mode	reverse mode
- evaluate the intermediate variable v_j and its directional derivative D_pv_j simultaneously	- does NOT evaluate v_j and D_pv_j simultaneously
- m passes	- 2 passes
- compute the gradient f with respect to the independent variables	- compute the sensitivity v_{j-m} of f with respect to the independent AND intermediate variable v_{j-m}
- evaluate the function from inside out	- traversing the computational graph backwards
- could use dual number	- can NOT use dual number
- have a larger algorithmic operation count (usually in a factor of 5)	- have to store the whole computational graph

Dual number

A dual number, similar to a complex number, has a real part and a dual part: $z=a+b\epsilon$, where ϵ is a high order term and we define $\epsilon^2=0$. Dual numbers are useful to encode the primal and the tangental traces. It is a useful data structure in carrying out the forward mode of autodifferentiation, since the function evaluation and directional derivative are calcualted simultaneously in the forward mode.For example, let f and g be two functions th $f\prime$ and $g\prime$ being their derivatives.We construct two dual numbers:

$$z_1 = f + f\prime\epsilon$$

$$z_2 = g + g\prime\epsilon$$

Therefore, we have:

$$z_1 + z_2 = (f + g) + (f\prime + g\prime)\epsilon$$

$$z_1 \cdot z_2 = (f \cdot g) + (f \cdot g\prime + g \cdot f\prime)\epsilon$$

It can be observed that adding dual numbers together resembels the addition both for the evluation and the directional direvative parts. Similarly, the multiplication of dual numbers resemble the multiplication of the functions in the real part and the product rule of the directional derivative in the dual part. Therefore, it is a useful structure to ncode the primal and the tangental traces.

Pomeranian Usage

The `Pomeranian` package will be distributed via PyPI or TestPyPI. To install `Pomeranian`, use an installer program with the following code

```
pip install Pomeranian
```

All the dependencies (`numpy`, `math`) wil also be installed.

After installation, you can simply import `Pomeranian`. Here is an example code of how to initialize a `ForwardMode` and `ReverseMode` object.

```
# import packages
import Pomeranian as ad

# functions
f1 = lambda x: ad.sqrt(ad.tan(x))
f2 = lambda x, y: ad.exp(ad.sin(x)-ad.cos(y)**2)
f_multi = [f1, f2]

# initialize forward mode
fw_AD = ad.ForwardMode(f_multi)
fw_AD.expression()
fw_AD.evaluate(x = 30, y = 60)
```

```
fw_AD.forward(x = 30, y = 60)

# initialize reverse mode
rv_AD = ad.ReverseMode(f_multi)
rv_AD.expression()
rv_AD.evaluate(x = 30, y = 60)
rv_AD.reverse(x = 30, y = 60)
```

See Implementation for more information about methods.

Software Organization

- **Directory structure**: The following is how we plan to organize the software.

```
main
├── .gitignore
├── LICENSE
├── README.md
├── setup.cfg
├── setup.py
|
├── .github/workflows
|   ├── coverage.yml
|   └── test.yml
|
├── docs
|   ├── milestone documents
|   ├── design documents
|   ├── pictures
|   └── ...
|
├── pomeranian
|   ├── __init__.py
|   ├── Automatic Differentiation (base class)
|   ├── Computational graph (class/module)
|   ├── Dual (class)
|   ├── Node (class)
|   ├── Forward Mode (class)
|   ├── Reverse Mode (class)
```

```
|   └── ...
|
├── tests
|   ├── Automatic Differentiation (base class)
|   ├── Computational graph (class/module)
|   ├── Dual (class)
|   ├── Node (class)
|   ├── Forward Mode (class)
|   ├── Reverse Mode (class)
|   └── ...
|
```

- **Modules**: There will be four directories in the software
    1. pomeranian: the main package containing all the modules/classes (including `ComputationalGraph`, `Dual`, `Node`, `ForwardMode`, `ReverseMode`, etc.) for AD; other dependencies (`numpy`, `math`) will also be imported to facilitate structure and calculation
    2. tests: unit tests for pomeranian AD algorithms and functions
    3. docs: additional documents and milstones for project development
    4. .github/workflow (hidden): workflow configuration files
- **Test suite design** Test suite will be included in the tests directory at the top level, which is the same level as the main package. Python built-in `unnittest` and `pytest` will be used to write test functions for each class separately. Github Actions will be used as a CI process (in Github Exterprise) to automatically test code interagraion and doucment generation.
- **Distribution & Considerations** As mentioned above, package will be distributed via PyPI or TestPyPI. Structure of the package is designed to be modular to optimize teamwork and collaboration, and minimize chances of merge conflicts.

Implementation

Core Data Structures

- Input:
    - Function
        - We will treat all functions as vector-valued. A scalar function will be a vector of length 1.
        - Each scalar function in the vector-valued function is represented by a string.
    - Differentiation point
        - Dictionary containing the variable name and value for each dimension of the point at which we perform AD
    - Seed vector: vector p for directional derivative

- Dual number: Class, for use in forward mode AD
- Node: Class, for use in reverse mode AD
- Intermediate values, partial derivatives to be stored in an np.arrays or tensors

Classes

- `Dual`: dual number, represent a number and the derivative of the function at the number, for use in forward mode AD
- `Node`: value, children, and associated local gradients, for use in reverse mode AD
- `AutoDiff`: base class for AD
- `ForwardMode`: implements forward mode AD
- `ReverseMode`: implements reverse mode AD
- `ComputationalGraph` (class/module): includes functions related to the conversion of the vector-valued input function into a computational graph

Method and Name Attributes

- `Dual` class: for use in ForwardMode
  - Attributes:
    - self.real: real part of dual number, to calculate value v_i of function
    - self.dual: dual part of dual number, to calculate value $D_p V_i$
  - Methods
    - Dunder methods (e.g. *_add_*, *_multiply_*, etc.)
    - Elementary operations: use operator overloading and scalar operations from numpy and math packages to define elementary functions on dual numbers
- `Node` class: for use in ReverseMode
  - Attributes:
    - self.value
    - self.children
    - self.partial_derivs
  - Methods:
    - Dunder methods (e.g. *_add_*, *_multiply_*, etc.)
    - Operator overloading to update values, local gradients, and children
- `AutoDiff` bass class:
  - Attributes:
    - self.function: vector-valued input function
  - Methods:

        - evaluate: return the value of function applied at input variables
        - dunder methods (e.g., *repr*, *str*, etc.)
- `ForwardMode` class (inherit AutoDiff):
    - Attributes:
        - self.function: inherited from AutoDiff class
    - Methods
        - forward: calculate derivative/jacobian of formula(s) by forward mode AD
        - dunder methods (e.g., *repr*, *str*, etc.)
- `ReverseMode` class (inherit AutoDiff):
    - Attributes:
        - self.function: inherited from AutoDiff class
    - Methods:
        - reverse: calculate derivative/jacobian of formula(s) by reverse mode AD
        - dunder methods (e.g., *repr*, *str*, etc.)
- `ComputationalGraph` class/module:
    - functions to output a visualization of the computational graph for users

External Dependencies

- `numpy` (data structure, elementary functions), `math` (elementary functions)

Elementary Functions

- Elementary functions will be definied in the `Dual` class for forward mode and `Node` class for reverse mode. We will use existing modules `math` and `numpy` define functions for dual numbers using operator overloading.
- Derivatives: Derivatives will be handled by the operator overloading. For forward mode, dervatives are caculated using the self.dual part of the dual number. For reverse mode, self.partial_derivs stores the partial derivative for a given Node instance with respect to each of its child nodes.

Implementation Example

We will implement operator overloading as in the following simple example. Consider the operator overloading for three-dimensional input x and $f(x)$ mapping from R^3 to R below.

$$\vec{x} = [x_0, x_1, x_2]$$

$$f(x) = x_0 + x_1 + x_2$$

```
def __add__(self, other):
    if isinstance(other, (int, float)):
        real_part = self.real + other
        dual_part = self.dual
    elif(isinstance(other, DualNumber)):
        real_part = self.real + other.real
        dual_part = self.dual + other.dual
    else:
        raise TypeError('Wrong type for addition')
    return DualNumber(real_part, dual_part)

>>> x_0 = Dual(1)  # Dual part initialized to 1 by default
>>> x_1 = Dual(2)
>>> x_2 = Dual(3)
>>> x = (x_0, x_1, x_2)  # define multi-dimensional x as a tuple
>>> f = x[0] + x[1] + x[2]
>>> f.real  # gives value at x = [1,2,3]
>>> f.dual  # gives deriv at x = [1,2,3]
```

Licensing

We license our program under an MIT License which is both simple and permissive. It allows any users to use, modify, or distribute the software without any associated liability for our team.

Feedback

Introduction(2/2):

This section was really nicely done!

Response: we left the section as is.

Background(2/2):

This section was really nicely done! Please make sure to check the incorrect math formula in your .md file

Response: we have fixed the math formula so it displays correctly.

How to use(3/3):

This section was really nicely done! Again those dollar signs make your file hard to read. Please update this part in your next submission.

Response: we have fixed the math formula so it displays correctly.

Software Organization(2/2):

Good

Response: we left the section as is.

Implementation(3.5/4):

You have included enough implementation details about this project. But I suggest you add a concrete example of how you will handle the operator overloading. For example in our class and assignment, it contains an example of the add operator of dual numbers. The goal of this milestone is to keep you on the right track of implementing your forward mode. Without a concrete example, it would be hard for us to justify the correctness of your design logic. When talking about the operator overloading, you also have to think about the input of high dimensional values.

Response: we have included a concrete example of how to handle operator overloading in the context of multidimensional data, through the __add__ dunder method with $X = [x_1, x_2, x_3]$.

License(2/2)

Total: (14.5/15)