Project Overview

Introduction ¶

Every computer program executes a sequence of elementary arithmetic operations and elementary functions. By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, and accurately, to working precision. This process is known as Automatic Differentiation.

Our group aims to develop a auto-differentiation API that supports forward and reverse mode. The importance of solving this problem stems from the widespread uses and applications of automatic differentiation, especially in scientific computing and science in general.

Essentially, the package inspects a sequence of elementary operations given as a Python function. The software then converts the sequence into a computational graph from which we can readily compute derivatives for arbitrary functions and points.

Background

Chain Rule

The underlying motivation of automatic differentiation is the Chain Rule that enables us to decompose a complex derivative of multiple functions into a sequence of operations using elementary functions with known derivatives. Below we present a sufficiently general formulation of the Chain Rule:

$$\nabla f_x = \sum_{i=1}^{n} \frac{\partial f}{\partial y_i} \nabla y_i(x)$$

We will first introduce the case of 1-D input and generalize it to multidimensional inputs.

One-dimensional (scalar) Input: Suppose we have a function $f(y(t))$ and we want to compute the derivative of f with respect to t. This derivative is given by:

$$\frac{df}{dt} = \frac{df}{dt}\left(y(t)\right)\frac{dy}{dt}$$

Multi-dimensional (vector) Inputs: Before discussing vector inputs, let's first take a look at the gradient operator ∇

That is, for $y \colon \mathbb{R}^n \to \mathbb{R}$, its gradient $\nabla y \colon \mathbb{R}^n \to \mathbb{R}^n$ is defined at the point $x = (x_1, \ldots, x_n)$ in n-dimensional space as the vector:

$$\nabla y(x) = \begin{bmatrix} \frac{\partial y}{\partial x_1}(x) \\ \vdots \\ \frac{\partial y}{\partial x_n}(x) \end{bmatrix}$$

We will introduce direction vector p later to retrieve the derivative with respect to each y_i.

Jacobian-vector Product

The Jacobian-vector product is equivalent to the tangent trace in direction p if we input the same direction vector p:

$D_p v = Jp$

Seed Vector

Seed vectors provide an efficient way to retrieve every element in a Jacobian matrix and also recover the full Jacobian in high dimensions.

Seed vectors often come into play when we want to find $\frac{\partial f_i}{\partial x_j}$, which corresponds to the i, j element of the Jacobian matrix. In high dimension automatic differentiation, we will apply seed vectors at the end of the evaluation trace where we have recursively calculated the explicit forms of tangent trace of f_i and then multiply each of them by the indicator vector p_j where the j-th element of the p vector is 1.

Evaluation (Forward) Trace

Definition: Suppose x = $\begin{bmatrix} x * 1 \\ \vdots \\ x_m \end{bmatrix}$, we defined $v * k - m = x_k$ for $k = 1, 2, \ldots, m$ in the evaluation trace.

Motivation: The evaluation trace introduces intermediate results $v\_k - m$ of elementary operations to track the differentiation.

Consider the function $f(x) : \mathbb{R}^2 \rightarrow \mathbb{R}$:

$f(x) = log(x_1) + sin(x_1 + x_2)$

We want to evaluate the gradient ∇f at the point $x = \begin{bmatrix} 7 \\ 4 \end{bmatrix}$. Computing the gradient manually:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \frac{1}{x_1} + \cos(x_1 + x_2) \\ \cos(x_1 + x_2) \end{bmatrix} = \begin{bmatrix} \frac{1}{7} + \cos(11) \\ \cos(11) \end{bmatrix}$$

Forward primal trace	Forward tangent trace	Pass with p = $[0, 1]^T$	Pass with p = $[1, 0]^T$
$v_{-1} = x_1$	p_1	1	0
$v_0 = x_2$	p_2	0	1
$v_1 = v_{-1} + v_0$	$D * pv * -1 + D_p v_0$	1	1
$v_2 = sin(v_1)$	$\cos(v_1) D_p v_1$	$\cos(11)$	$\cos(11)$
$v_3 = log(v_{-1})$	$\frac{1}{v*-1} D_p v * -1$	$\frac{1}{7}$	0
$v\_4 = v_3 + v_2$	$D * pv * 3 + D_p v_2$	$\frac{1}{7} + \cos(11)$	$\cos(11)$

$$D_p v_{-1} = \nabla v_{-1}^T p = (\frac{\partial v_{-1}}{\partial x_1} \nabla x_1)^T p = (\nabla x_1)^T p = p_1$$

$$D_p v_0 = \nabla v_0^T p = (\frac{\partial v_0}{\partial x_2} \nabla x_2)^T p = (\nabla x_2)^T p = p_2$$

$$D_p v_1 = \nabla v_1^T p = (\frac{\partial v_1}{\partial v_{-1}} \nabla v_{-1} + \frac{\partial v_1}{\partial v_0} \nabla v_0)^T p = (\nabla v_{-1} + \nabla v_0)^T p = D_p v_{-1} + D_p v_0$$

$$D_p v_2 = \nabla v_2^T p = (\frac{\partial v_2}{\partial v_1} \nabla v_1)^T p = \cos(v_1)(\nabla v_1)^T p = \cos(v_1) D_p v_1$$

$$D_p v_3 = \nabla v_3^T p = (\frac{\partial v_3}{\partial v_{-1}} \nabla v_{-1})^T p = \frac{1}{v_{-1}} (\nabla v_{-1})^T p = \frac{1}{v_{-1}} D_p v_{-1}$$

$$D_p v_4 = \nabla v_4^T p = (\frac{\partial v_4}{\partial v_3} \nabla v_3 + \frac{\partial v_4}{\partial v_2} \nabla v_2)^T p = (\nabla v_3 + \nabla v_2)^T p = D_p v_3 + D_p v_2$$

Computing the Derivative

Generalizing our findings:

From the table, we retrieved a pattern as below:

$$D * pv_j = (\nabla v_j)^T p = (\sum *i < j \frac{\partial v * j}{\partial v_i} \nabla v_i)^T p = \sum *i < j \frac{\partial v * j}{\partial v_i} (\nabla v_i)^T p = \sum *i < j \frac{\partial v_j}{\partial v_i} D_p v_i$$

Reverse Mode

The mechanism of reverse mode is defined as the following:

Step 1: Calculate $\frac{\partial f}{\partial v_j}$

Step 2: Calculate $\frac{\partial v\_j}{\partial v_i}$ where v_i is the immediate predecessor of v_j

Step 3: Multiply the result obtained in step 1 and step 2, which results in the following: $\frac{\partial f}{\partial v*j}\frac{\partial v*j}{\partial v_i}$

How to Use

There will be a simple public interface for our end users. Users will import a module `functions` which contains familiar elementary functions (*sin, cos, exp, log, etc.*) and a class to support creation of custom user defined functions. Other modules for support of dual numbers and computation graphs will also be included.

Example usage:

```
from autodiff.functions import Function, sin, cos, exp

func = Function(
  lambda x: x[1] * f.sin(x[0]), lambda x: f.exp(f.cos(x[2])),
  input_dim=3
)

x = np.array([1, 2, 3])
func.derivative(x, mode='reverse')
```

Dual Numbers

```
from autodiff.numbers import Dual
x = Dual(2, 1)
a = 1
b = 2
c = 3
f = a*x + b*x + c*x
print(f.real)
print(f.dual)
```

Further detailed descriptions for classes, methods, and basic operations will be documented in our API reference

Software Organization

- The directory structure will look similar to

```
📦 repo
┣ 📁 .github
┃ ┗ 📁 workflows
┃ ┃ ┣ 📜 cd.yml
┃ ┃ ┗ 📜 ci.yml
┣ 📁 .venv
┣ 📁 autodiff
┃ ┣ 📜 __init__.py
┃ ┣ 📜 functions.py
┃ ┣ 📜 graph.py
┃ ┗ 📜 numbers.py
┣ 📁 test
┃ ┣ 📁 fixtures
┃ ┣ 📁 test_functions
┃ ┣ 📁 test_graph
┃ ┣ 📁 test_numbers
┃ ┗ 📜 __init__.py
┣ 📜 .devcontainer
┣ 📜 .gitignore
┣ 📜 Dockerfile
┣ 📜 LICENSE
┣ 📜 README.md
┣ 📜 poetry.toml
┗ 📜 pyproject.toml
```

Testing

- We will be using a test driven development process
- Test suite will reside locally in the repository `/test` directory
- Test execution will be incorporated into our CI workflow

Distribution

- Package will be distributed via PyPI using https://test.pypi.org/ (https://test.pypi.org/) for our demo deploys and https://pypi.org (https://pypi.org) for production deploys
- We will use github actions for our CD deploys. Demo deploys will be automatic on pushes to main. Production deploys will be based on version tags.

Repository Management

- We will enforce branch protection rules on our default branch `main`
- Key Protection Rules
  - Require a pull request before merging
  - Require status checks to pass before merging
  - Require approvals
  - Require linear history

Other Considerations

- Packaging and dependency management will be handled via Poetry (https://python-poetry.org/)
- Docker devcontainers will be available for project collaborators
- Source code will adhere to The Black code style (https://black.readthedocs.io/en/stable/the_black_code_style/index.html) which is PEP 8 compliant

Implementation

What classes do you need and what will you implement first?:

- Node : foundation of our computational graphs
  - parents : parent node(s)
  - children : child node(s)
- Graph: computational graph ; this is our core data structure
  - root_nodes : root node(s) of the graph
  - tail_nodes : tail node(s) of the graph (for reverse mode)
- Function : creates user defined functions
  - graph : computational graph to compute derivative
  - derivative : returns derivate at given input using forward or reverse mode (specified via key word argument)
  - Our function class will handle multi-output functions using multiple Python function inputs (see How to Use above)

- Dual : supports creation and computation of dual numbers
    - We will implement a variety of dunder methods for basic operations (addition, subtraction,...)

In addition, we will implement a suite of elementary functions with known derivatives which will natively integrate with our Function and Graph classes. We will rely on our one dependency -- Numpy -- for computation, but incorporate methods to merge computations into a given function's graph.

Licensing

We will use an open source license, specifically the MIT License. This license places minimal restrictions on our end users and enables them to freely use and improve upon our project. Additionally, this license minimizes legal obligations for our team.