`cyanDiff` – a forward-mode AD library

Introduction

`cyanDiff` will implement forward-mode automatic differentiation in Python using Dual Numbers to differentiate many functions.

Solving the automatic differentiation problem is useful not only because it is a core mathematical operation that can be used in scenarios like CASes (computer algebra systems), but because it has key applications in tasks such as machine learning and data analysis. In particular, automatic differentiation is a critical component of the ML model training process. By modeling a generalized differentation scheme within `cyanDiff`, users will be able to differentiate all sorts of functions composed from basic elementary functions and variables, which is exactly the type of thing that appears in complex ML models with heterogeneous architectures.

Background

Automatic differentiation and our package rely on the chain rule in calculus. For scalars, the chain rule is given by:

$$\frac{d}{dx}(f \circ g)(x) = f'(g(x)) \cdot g'(x).$$

For example, if we are differentiating the elementary function, $\sin x^2$, we would get $2x \cdot \cos x^2$ because cos is the derivative of sin and $2x$ is the derivative of x^2.

At its core, automatic differentiation allows for a program to differentiate an elementary function (or the composition of elementary functions, including exp, trig functions, polynomials, log etc) by repeatedly applying the chain rule to the compositions of functions. AD also keeps track of the directional derivative, which is given by multiplying the currently computed derivative by a direction vector commonly denoted p.

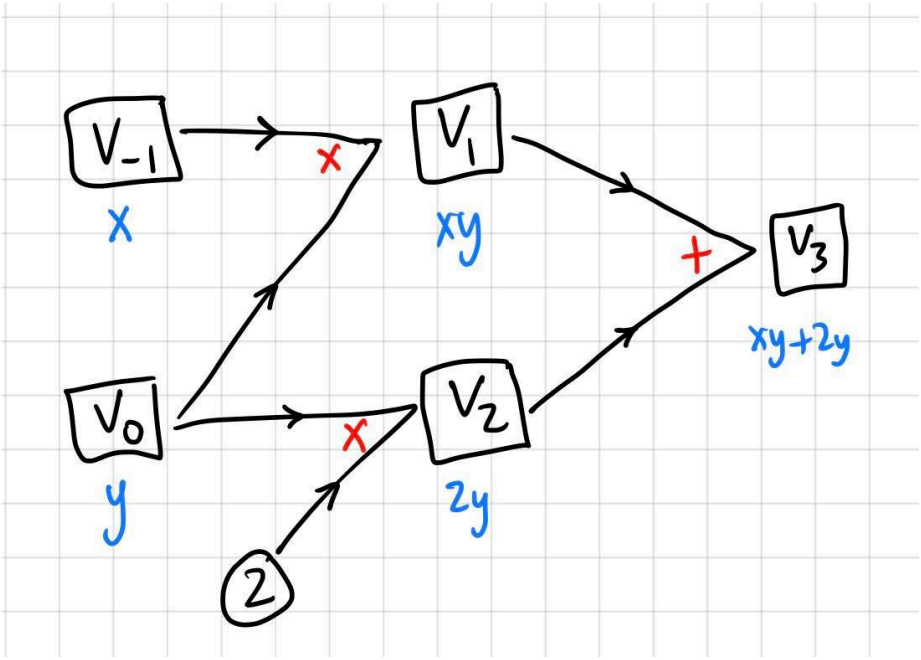
This necessitates the use of a computational graph. Each node in the graph keeps track of the derivative and the directional derivative. Then, edges between the nodes are represented by functions (addition, multiplication, e^x etc). It is worth noting that while we gave scalar functions above, we will support the differentiation of elementary vector valued functions $f : \mathbb{R}^m \to \mathbb{R}^n$.

Now we describe the technical details about implementing automatic differentiation. Specifically, we talk a bit about dual numbers and how they are used in the automatic differentiation process. Dual numbers are similar to complex numbers, except instead of using the imaginary unit i, we use a value ε such that $\varepsilon^2 = 0$. So, dual numbers are of the form $a + b\varepsilon$, where a, b are real numbers.

What is the advantage of using dual numbers? There is a clever way of using the properties of ε so that we can store both the primal and tangent traces of

a computation within a *single* dual number. This drastically simplifies computations, and combined with overloaded DualNumber-supporting functions, forward-mode AD becomes a simple task.

To understand how DualNumbers can be used to streamline AD, we first discuss the idea of a computational graph. All mathematical functions can be represented in terms of simpler "elementary" functions. These functions are things such as addition, exponentiation, and cosine. We then represent a mathematical function as a composition of a bunch of these elementary functions, on top of a set of input "independent" variables. Let's consider the function $f(x, y) = x * y + 2y$ for the remainder of this section. This function takes two input variables x, y and is computed as follows, in terms of elementary functions: we multiply x and y, multiply 2 and y, and then add those two results together in order to get $f(x, y)$. We can then represent these chained operations in the following graph, where the v_i represent intermediate computed values when $i > 0$, and for $i < 0$ we simply have the independent input variables:



Now, suppose we want to use this computational graph to help us calculate the gradient of v_3, the function value, with respect to the inputs v_{-1} and v_0. The trick is to store both the imtermediate value and (directional) derivative in the form of a single Dual number at every node of the computational graph. So, suppose we want to compute the directional derivative in a direction vector p at some point $(x_0, y_0) = \mathbf{x}$. We would store at every node v_i the value $v_i(\mathbf{x}) + D_p(v_i)(\mathbf{x})\varepsilon$, where D_p denotes the directional derivative in direction p (the Jacobian multiplied by p).

Then, by storing both the function value $v_i(\mathbf{x})$ and the derivative $D_p(v_i)(\mathbf{x})$ in a single Dual number, we can compute successive function values and derivatives simply by performing the corresponding elementary function operations on the dual numbers themselves. For example: suppose we're in the above graph and want to compute v_3's node from v_1 and v_2. We simply add their Dual number values to get $v_1(\mathbf{x})+v_2(\mathbf{x})+\left[D_p(v_1)(\mathbf{x}) + D_p(v_2)(\mathbf{x})\right]\varepsilon$, which is exactly $v_3(\mathbf{x}) + D_p(v_3)(\mathbf{x})\varepsilon$, by the addition rule for derivatives.

This holds true for more complex scenarios; suppose we wanted to multiply nodes v_1 and v_2. The product rule is required when computing the derivative of the resultant node $v_3 = v_1 \cdot v_2$. But when using Dual numbers, simply multiplying the Dual number representations works: $(v_1(\mathbf{x}) + D_p(v_1)(\mathbf{x})\varepsilon) \cdot (v_2(\mathbf{x}) + D_p(v_2)(\mathbf{x})\varepsilon) = v_1(\mathbf{x})v_2(\mathbf{x}) + \left[v_1 D_p(v_2)(\mathbf{x}) + v_2 D_p(v_1)(\mathbf{x})\right]\varepsilon$, which is exactly what we'd get from the product rule. Notice how the term with ε^2 cancels out by definition.

How to use `cyanDiff`

Similar to many existing AD packages, `cyanDiff` will allow the user to instantiate variables symbolically, and then combine those variables along with other operators to form functions. For example:

```
import cyanDiff as cd

x, y, z = cd.var("x y z")
f = 2*x + y**3 + x*z + 4*(x**2)*(y**4)
```

Other elementary functions can be imported from our package as well, which are overloaded versions of the usual functions from NumPy that now support Dual numbers.

```
f_2 = 2 * cd.sin(x) + cd.exp(y)
```

Besides simple real-valued functions, vector functions can also constructed. For example (continuing from above), the following creates a function $g : \mathbb{R}^2 \to \mathbb{R}^3$:

```
g_1 = x + 100*y
g_2 = x*y + y**2
g_3 = x / (y + 3)
g = cd.matrix(g_1, g_2, g_3)
```

Once a function has been composed via mathematical operators and variables, we can compute its derivative at a specified point:

```
point1 = (1, 2, 3)
res1 = f.diff_at(point1)

point2 = (4, 5)
res2 = g.diff_at(point2)
```

The output of `diff_at` is a matrix, specifically the Jacobian of the specific function at the specified point. For example, `res1` is the Jacobian of the function `f` evaluated at `point1`. Under the hood, forward-mode AD is making multiple passes per basis vector direction and stitching together the requisite directional derivatives in order to form the full Jacobian.

Special instances of the derivative, notably the gradient, can be computed via corresponding functions:

```
point3 = (6, 7)
res3 = g.grad_at(point3)
```

Under the hood, the `grad_at` function is simply computing the transpose of the Jacobian at the specified point. Again, the output of `grad_at` is a matrix.

This package requires the following dependencies to be installed: `numpy`

Software Organization

```
AutomaticDifferentiation/
|----__init__.py
|----DualNumber.py
|----Node.py
|----AD.py
|----tst/
     |----__init__.py
```

- The directory structure will consist of a parent directory called `AutomaticDifferentiation`, and there will be modules within this directory including `DualNumber`, `Node`, an `AD` module, and a `tst` subdirectory for running tests.
- For modules, we plan to include and import `DualNumber` and `Node`. `DualNumber:`

  `Node:`
- Our test suite will live as a separate subdirectory in the same directory as our `AD` module and associated modules like `DualNumber` and `Node`. This will allow us to easily import our `AD` module and run tests, where we can compare an analytically comptued derivative to our AD program's results and ensure that the results match.
- We will distribute `cyanDiff` via PyPI, with all the necessary package configuration files. The package can thus be installed via

```
python3 -m pip install cyanDiff
```

Implementation

The classes we will need are a `DiffGraph` class for the graph structure, a `DiffNode` class for the nodes in the auto-differentiation graph, and a `DualNum` class for storing dual numbers. Our key data structure will be a graph, included in the `DiffGraph` class, which will be made up of individual node data structures. We will also make use of array data structures (in particular `numpy` arrays), and as said earlier, we will create a class for dual numbers with each `DualNum` object having two attributes for the two parts of the dual number. We will incorporate dual numbers by implementing our `DualNum` class in such a way that its operators are defined to be compatible with the calculations performed using methods from our other classes.

- **DualNum**

  The `DualNum` object has two attributes for the two parts of the dual number. `real` representing the primal trace value and `dual` representing derivative (tangent trace). The method `DualMath`in the class can convert basic math operations to dual number format.

We would implement the `DualNum` class first as it is the lowest level class here.

As done in HW 4, we need to define some functions. These include a constructor, dunder add, dunder mul, dunder radd and dunder rmul. This allows for adding and multiplying Dual Numbers with other Dual Numbers along with ints and floats in either order.

For example, here is what our constructor could look like:

```
def __init__(self, real, dual = 1):
  if isinstance(real, int) == False and isinstance(real, float) == False:
    raise TypeError

  if isinstance(dual, int) == False and isinstance(dual, float) == False:
    raise TypeError

  self.real = real
  self.dual = dual
```

We implement overloading in case user-defined math functions ever use a constant `int` or `float`. We define our basic math operations to handle when one of the values is an `int` or a `float` number. For example, when adding an `int` or a `float` we add the value to the real part of the dual number, and when multiplying we multiply both the real and dual parts of the dual number by the `int` or `float`. We include the examples below to show how this works.

```
a_0 = 1
z_1 = DualNumber(5,5)
```

```
res1 = z_1 + a_0

a_1 = 2.0
z_2 = DualNumber(2,2)
res2 = a_1 * z_2

print(res1.real)
print(res1.dual)
print(res2.real)
print(res2.dual)
```

```
6
5
4.0
4.0
```

We include an example implementation of `__add__` and `__radd__` for our dual number class that accounts for integers and floats:

```
def __add__(self, other):
    if isinstance(other, DualNumber):
        return DualNumber(self.real + other.real, self.dual + other.dual)
    elif isinstance(other, (int, float)):
        return DualNumber(self.real + other.real, self.dual)
    else:
        raise TypeError("Must be 'int,' 'float,' or 'DualNumber'")

def __radd__(self, other):
    return self.__add__(other)
```

When implementing forward mode AD, we will not use any graph and node structure. These sections are kept below as considerations for implementing reverse mode AD.

For forward mode AD, we will implement our own overloaded primitives, e.g. the basic arithmetic operations, trig functions, and exponentiations, with which users can define math functions of one or multiple dimensions using our CyanDiff instantiated variables. We use our `diff_at` function, which will take one of these user-defined math functions as well as a user provided point to then evaluate the derivative at the given point.

In particular, we will take the user defined point and convert each coordinate to its own dual number. The real part is the same as the real value given in the user-inputted point. The dual part depends on the value of the p directional vector. For n-dimensional user input, we will calculate the primal and tangent traces n times, with n different p vectors, with each p vector being a different standard basis vector (one position has value 1 and the rest are all 0). We then provide these dual numbers as input to the math function defined in terms of our dual-number compatible functions, which will give us a dual number or a

vector of dual numbers as the output. By the properties of dual numbers as described in the background section, the real part of the dual output will be the primal trace, and the dual part will give the tangent trace. We then take the tangent trace for each p vector and use them to construct our Jacobian.

- **DiffNode**

Recall as above that this only for reverse mode.

In `DiffNode` class, we define a node-type object for each of the points in our computational graph, which we must store when performing reverse mode AD. Each node will have to have a children field, the value at that node, and the associated basic operation or function (e.g. trig functions or logarithm) associated with that node. The children field will consist of a list of tuples of child node id and the corresponding partial derivative of that child with respect to the parent

- **DiffGraph**

Our `DiffGraph` class will have an attribute that is a list of nodes of our `DiffNode` type. For reverse mode, we will use this to store the computational graph.

Licensing

We choose the MIT license because it is a relatively simple and permissive license, allowing other people to freely use and modify our code, as well as distribute a closed source version, while protecting us from developers claiming liability. The only package we plan to use in `numpy`, which is licensed under a liberal BSD license, so we are free to use the MIT license in our project.

Feedback

Here we address the feedback we received for milestone 1. The comments are in bold and our responses are in normal text.

Background

It would be helpful if you include more background of Dual Number and why it is useful in your package. - I would recommend you to have some graphs to explain the computational graph. Remember the users of your package may be not familiar with the technical terms.

We addressed these suggestions by including a discussion of how Dual Numbers, by their properties under basic operations, effectively allow us to track the primal and tangent traces in forward mode AD. Thus, they are quite useful in our package. We also include a computational graph figure to help with reader understanding.

How to Use

It is not so clear about the difference between the usage of grad_at and diff_at (-0.25)

We specify that the gradient (what `grad_at` returns) is the transpose of the Jacobian (which is what `diff_at` returns). In short, their return values are matrices which are transposes of each other.

Software organization

Great design! But hope the relationship between different modules can be more clear. Maybe draw a structure graph to show it.

We include a structure graph to show the relationship between the different modules, and further clarify in writing the module structure.

Implementation (explicit design considerations)

Please be clear how DualNumber class will be implemented. Showing some example code will be preferred. (-0.25)

We lay out the attributes and methods that the `DualNumber` class will have. We provide example code for the implementation of some of the basic operators for dual numbers, and examples of how they will function.

Do you really need to store all the parent and corresponding functions if you use the forward mode? Think more about the design of your Node class. (-0.5)

We no longer use a `Node` class for our implementation of forward mode (though we keep the idea around for our consideration of how to implement reverse mode). We instead simply allow the user to define math functions using basic math operators and functions provided from our package, and have them be passed as input for differentiation. We do not store parents and corresponding functions any longer, and instead use the properties of dual numbers in order to record our primal and tangent trace calculations.

Where would you put the elementary functions? How would you overload it? Think more about the design of elementary functions.

We implement our elementary functions in our `DualNumber` class. We overload our elementary functions to accept `int` and `float` as well as dual numbers, so that the user can include `int` and `float` values in their defined math functions that they pass for differentiation. This is illustrated in the example code that we provide.