Introduction

Differential calculus was invented as a formal branch of mathematics in the middle of the 17th century independently by Isaac Newton and Gottfried Leibniz. Newton's orginal term for a derivative with respect to time was a "fluxion," which gave its name to his last book, *Method of Fluxions*, published posthumously. Newton used differential calculus to solve the problem of the motion of the planets around the sun, and it has proven to be an essential tool for the sciences ever since. In the modern era, essentially all scientific calculations of interest are performed on computers. There are many scenarios where we know how to compute a function of interest and would like to efficiently evaluate its derivative(s). The canonical example is root finding. If we know a function's derivative, we can iteratively improve from a starting guess until we find a root using a simple procedure, Newton's Method. Many phenomena of interest in physics and other sciences can be described as differential equations (either ODEs or PDEs). The ability to efficiently evaluate derivatives is crucial in numerically solving these systems. In recent years, Machine Learning has been a hot research area. Solving ML problems often hinges on the ability to train a neural network using some form of a gradient descent algorithm. As the name suggests, this algorithm requires the ability to evaluate not just the function (here the output of the neural net) but also its first derivative, which in practice is typically the change in the error metric with respect to the parameter values. These neural networks are massively complex functions that can have millions of parameters. A procedure of numerically differentiating the network by shifting the value of each parameter has cost that scales linearly with the number of parameters. Some form of automatic differentiation is vital to the practical use of neural nets. One of the most successful machine learning libraries is TensorFlow by Google, which at its core is an enterprise class automatic differentiation library.

Background

Calculus gives us a few simple rules we can apply to compute the derivative of a function that is the result of a combination of two more simple functions. Calculus rules include the sum, product, and quotient of two functions. The most important calculus rule in Automatic Differentiation is the Chain Rule of calculus. This states that the derivative of a composition is the product of two derivatives. $f'(u(x)) = f'(u(x) \cdot u'(x)$ The chain rule works in multiple dimensions. If f is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$, its derivative an m x n matric called the Jacobian. The chain rule in the multidimensional case tells us to take the matrix product of an m x r matrix and an r x n matrix to compute the derivative.

The essential idea of Automatic Differentiation is that any computation performed by a computer will consist of a set of basic steps, e.g. function evaluations. These may be strung together in a complex graph with multiple steps, but each step will be a basic operation. For the evaluation of a mathematical function, each step will consist of either a "primitive" or built-in function (e.g. +, -, x, /, sqrt, log, exp, pow, etc.) or a composition of these primitives. As long as we know how to evaluate both the function f(x) and its derivative f'(x) at each step, we can trace the calculation through the

computation graph and also keep track of the first derivative. This is the idea at the heart of Automatic Differentiation.

The forward mode of Automatic Differentiation takes 3 arguments, a function, $f(x)$, a vector at which to evaluate the function, $x \in \mathbb{R}^n$, and a vector of 'seed' values for the derivatives, $dx \in \mathbb{R}^n$. Following the standard order of operations, the innermost elementary function calls of f are evaluated at x and dx. As above, as long as we know how to evaluate both the elementary function f(x) and its derivative f'(x) at each step, pass the results of each to enclosing functions and apply the chain rule to keep track of the first derivative. Once all function calls are complete, the algorithm returns the values $f(x)$ and $f'(x)$.

For example, if we wish to use the forward mode algorithm to differentiate $f(x) = e^{x_1} cos(2x_1 + x_2)$ at the point $x = (x_1, x_2) = (0, \pi/2)$, and we were only interested in the derivative with respect to x_1 we would pass the function f, the vector x, and $dx = (1, 0)$. Starting from the innermost functions, we evaluate the derivative and the value of e^{x_1}. The value is the scalar 1 and the derivative is a vector $(1, 0)$. Moving on, we evaluate the * operator on $2x_1$, giving a value 0 and the vector (2,0). The + operator gives the value $\pi/2$ and the vector (2,0). Cosine gives the scalar 0 and the vector $(-sin(2), 0)$. Finally, the multiplication operator between e and cos gives $f(x) = 0$ and $f'(x) = (-sin(2), 0)$ which would be the output. Because we passed the seed value $dx = (1, 0)$, $f'(x)$ has a zero in the place corresponing to x_2. The zero in the seed carries through to the final derivative, only returning the non-zero entries.

How to Use Fluxions

The intention is that users will interact with the Fluxions module in a way that is very similar to the way they interact with `numpy`. The module will implement a complete set of elementary functions that can be strung together to form more complex expressions by passing them as arguments to any other function objects. Operator overloading will also enable users to connect function objects using symbolic operators such as `+`, `-`, `*`, `/`, and `**`.

Symbolic variables are themselves function objects, which can be set to take on one or more values at the time they are instantiated or after a larger expression has been built up. For instance, if a user wanted a differentiable version of the standard normal distribution, they could write either of the following:

```
import fluxions as fl

std_norm = fl.exp(fl.power(fl.var('z'),2))

z = fl.var('z')
std_norm = fl.exp(z**2)
```

Here, `exp`, `power`, and `var` are all instances of classes in Fluxions that implement the concept of a differentiable function. The command `fl.var('z')` creates an instance of the `variable` class, which is very lightweight and essentially just keeps track of the variable as a named entity in the computation graph.

All function classes implement a "fluxion" interface by defining two methods, `eval` and `diff`, which respectively evaluate and differentiate the function at one or more specified values. The user may evaluate a function (or composition of functions) and/or its derivative either by associating values with a `var` object, or by evaluating the overall functional expression (i.e. composition of functions) after it has been constructed and associated with a single function object. For instance, suppose the user wanted to evaluate the standard normal and its derivative at a set of points `z = np.linspace(-6, 6, 1201)` with seed values `dz = np.ones_like(z)`. The following are equivalent ways to accomplish this with the Fluxions module using the variables-passed-as-dictionary syntax:

```
z  = np.linspace(-6, 6, 1201)
dz = np.ones_like(z)

## 1st option
y, dydz = fl.exp(fl.var({'z':z, 'dz':dz})**2)


## 2nd option
# build up function expression first
std_norm = fl.exp(fl.var('z')**2)

# then evaluate function and derivative at specified values
y = std_norm.eval({'z':z})
dydz = std_norm.diff({'z':z, 'dz':dz})
```

These ideas can also be extended to functions of multiple variables. For example, to differentiate the function,

$$f(x,y,z) = \frac{1}{xyz} + \sin\left(\frac{1}{x} + \frac{1}{y} + \frac{1}{z}\right)$$

at the point $a = (1,2,3)$, the user could write:

```
x = fl.var('x')
y = fl.var('y')
z = fl.var('z')
f = 1 / (x*y*z) + fl.sin(1/x + 1/y + 1/z)
a = np.array([1, 2, 3])
val, der = f.jacobian(a)
```

The above expression using the variables-passed-in-order syntax, where the first value in the array will be interpreted as `x`, the second as `y`, and so on in order of appearance in the function. Behind the scenes, the expression `(x*y*z)` would be evaluated as `product(product(x, y), z)`, while `1/(x*y*z)` would be treated as `quotient(const(1), product(product(x, y), z))`, where the implicit promotion from the integer `1` to a constant float would be handled in the appropriate operator overload call, here `__rdiv__`.

The method `f.jacobian` would compute the Jacobian matrix of the function. In this case, since f goes from $\mathbb{R}^3$ to $\mathbb{R}^1$, the Jacobian will be a 1x3 matrix.

`fluxion` objects (differentiable functions) will be general enough to handle functions from $\mathbb{R}^n$ to $\mathbb{R}^m$. Operations in `fluxions` will be vectorized to the fullest extent possible by building the module on top of the numpy package. In order to simultaneously evaluate T inputs in $\mathbb{R}^n$, the user will pass as an argument an array with shape (T, n). If the user requests a derivative evaluated at one seed points with the `diff` method, the return arrays will be the function evaluation `val` with shape (T, m) and the derivative `der` also with shape (T, m). If the user requests multiple specific points with the method `diff`, the return for `der` would instead be an array of shape (T, m, k) where k is the number of seed points given. If the user requests the full Jacobian, they will receive an array `der` with shape (T, m, n). We may add an optional flag `squeeze` that defaults to `False` that would squeeze out array indices of size 1, since that is frequently the desired behavior from users.

Software Organization

The fluxions library will have the following directory structure:

```
fluxions/
      build/
            lib/
            ...
      fluxions/
            __init__.py
            fluxions.py
            tests/
                  __init__.py
                  test_integration.py
                  test_module1.py
                  test_module2.py
                  ...
      fluxions_pkg.egg-info/
            dependency_links.txt
            PKG-INFO
            SOURCES.txt
            top_level.txt
```

```
            dist/
                  fluxions_pkg-0.0.1-py3-none-any.whl
                  fluxions_pkg-0.0.1.tar.gz
            .gitignore
            .travis.yml
            LICENSE
            README.md
            setup.cfg
            setup.py
```

The main module for the library will be `fluxions.py`. This will include the implementation for the workhorse `fluxion` class, which embodies the concept of a differential function from $\mathbb{R}^n$ to $\mathbb{R}^m$.

We are using the 'tests as part of application code' framwork from the `pytest` documentation. The test suite will be in its own modules. Each code module will have one test module containing unit tests for that module. In addition, we may develop some integration style tests to assess the overall system. We plan to use `TravisCI` for continuous integration testing and `Coveralls` for code coverage.

We plan to distribute this package using `PyPI`. The above directory structure shows what the directory will look like after it has been packaged up for distribution. The contents of the `build/` directory will by system specific (macosx, linux, etc.).

Implementation

The primary consideration driving our decisions about data structures will be efficient vectorization. This is a library that needs to operate smoothly with `numpy` and allow high performance calculations of derivatives on large arrays. The goal is that the time taken to evaluate the derivative of a function on a large array with `fluxions` should be *comparable* to the time required to evaluate the closed form derivative. There is not an expectation it will be as fast as the analytical derivative, but it should not be orders of magnitude slower either. This means we will not be using e.g. python `list` instances for large amounts of numerical data, and we won't have for loops over e.g. all T samples of a set of inputs passed in as a `numpy.ndarray`.

The basic implementation plan for forward mode differentiation is *not* to build a calculation graph in the style of e.g. Google TensorFlow. Instead, the `fluxion` class will have methods to return function evaluations and derivatives. When a `fluxion` is built up as a combination of smaller `fluxion` instances using supported mathematical operations, the derivative of the composition will be computed in situ using the rules of calculus. So if `u` and `v` are both `fluxion` instances depending on variable x, and the library is evaluating the derivative of `f = u * v`, it perform intermediate calculations that look like this in a stylized sense:

```
u_val, u_der = u.forward_diff(input, seed)
v_val, v_der = v.forward_diff(input, seed)
f_val = u_val * v_val
f_der = u * v_der + u_der * v
```

This is just the familiar product rule from high school calculus. The first important consideration is that rather than using a computation graph, we will rely on python's engine for parsing and evaluating expressions. This will combine all the elements in a complex differentiable function in the right order. All we need to do is define the elementary functions and the rules for combining them, of which the product rule above is one example. The second important consideration is that the steps required to compute the derivative of `f` in terms of the evaluations and derivatives of `u` and `v` are all natively vectorizable in `numpy`. As long as we keep careful track of the shapes of our arrays, we will have efficiently vectorized calculations without any for loops or data structures less efficient than `numpy` arrays.

Most `fluxion` class instances will be created by using the built-in elementary functions and combining them. However we will also create a constructor that allows a user to construct a `fluxion` by providing Python functions for both `f(x)` and `f_prime(x)`, as follows:

`fluxion(eval_func, der_func)`

So as an example, a user could write `my_sin = fluxion(np.sin, np.cos)` and get back an instance that behaved identically to `fluxion.sin`. When a user defines a fluxion this way, all responsibility will rest with the user to ensure the `der_func` really is the exact derivative of `eval_func`! On the other hand, the library developers (us) will assume this responsibility for all the provided functions.

Here are some preliminary methods for the `fluxion` class:

```
set_var_order(var_names)
diff_fwd(input, seed)
diffs_fwd(input, seeds)
diff_num(input, seed)
diffs_num(input, seed)
jacobian_fwd(input)
jacobian_num(input)
hessian_fwd(input)
hessian_num(input)
```

We would like to keep the external dependencies to a minimum to limit the complexity of the library. The current plan is to limit the dependencies on calculations to `numpy`. We will have additional dependencies for testing and packaging. All elementary functions will be delegated to their `numpy` equivalents. We will incorporate all the commonly used differentiable mathematical functions in `numpy`. A complete list of mathematical functions on numpy can be found here:

https://docs.scipy.org/doc/numpy-1.15.0/reference/routines.math.html

The most important functions on this list will include: `add, subtract, multiply, divide, power, sqrt, exp, log, sin, cos, tan, arcsin, arccos, arctan, arctan2, sinh, cosh, tanh, arcsinh, arccosh, arctanh`.