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 used it to solve the problem of the motion of the planets around the sun, and it's 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 rest as they say isbookkeeping...

How to Use Fluxions

The intention is that users will interact with the Fluxions library in a way that is very similar to the way they interact with `numpy` . Basic functions such as `exp` would be imported from the module anagously to `numpy`, e.g. `import fluxions as fl` Suppose a user wanted a differentiable version of the standard normal distribution. They could write

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

Here `fl.exp` would be an instance of a class in Fluxions that implements the concept of a differentiable function, as would `fl.power`. The command `fl.var('z')` would identify `z` as a formal parameter for the libary. Before it evaluates the expression, it would create a set of all the distinct variables that appear. The library would use the chain rule to evaluate the function and its derivative on the required inputs. Suppose the user wanted to evaluate the standard normal on a set of points `z = np.linspace(-6, 6, 1201)` with seed values `dz = np.ones_like(z)`. They could invoke the forward mode with the command

```
p, dp_dz = std_norm.diff_fwd(input=z, seed=dp_dz)
```

Basic mathematical operations will be expressible using operator overloading for `+`, `-`, `*`, `/`, and `**`. So for example, a user should also be able to construct an equivalent standard normal via

```
std_norm2 = fl.exp(fl.var('z')**2)
std_norm3 = fl.exp(fl.var('z')*fl.var('z))
```

On homework 4, problem 1 asked us to differentiate at the point $a = (1, 2, 3)$.

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

Here is how a user might solve this using the fluxions library:

```
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)
f.set_var_order(['x', 'y', 'z'])
a = np.array([1, 2, 3])
val, der = f.jacobian_fwd(input=np.array([1, 2, 3]))
```

The first three statements would create instances of a `variable` class. This would be very lightweight and essentially just keep track of the variable as a named entity in the computation graph. The fourth statements is where all action would take place. It would create an instance of the `fluxion` class, the workhorse for differentiable functions in this library. Behind the scenes, the library would evaluate the expression `(x*y*z)` as `product(product(x, y), z)`. It would treat `1/(x*y*z)` 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 command `f.set_var_order` informs the library that when it sees arguments as an array with 3 columns (matching the number of inputs), it should bind them to the variables x, y, z in that order. 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.

One important set of conventions would relate to the shape of arrays. `fluxion` objects (differentiable functions) will be general enough to handle functions from $\mathbb{R^n}$ to $\mathbb{R}^m$. (This convention is common and the mnemonic to recall it is that the Jacobian will be an m x n matrix.) Operations in `fluxions` will be vectorized to the fullest extent possible — no nested for loops for calculations on arrays unless the user is a noob! In order to submit T inputs in $\mathbb{R}^n$, the user will pass as input an array with shape (T, n). If the user requests a derivative evaluated at one seed points with the `diff_fwd` 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 `diffs_fwd`, 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.

To the extent possible, if `numpy` has made a decision regarding the interface of a mathematical function, we will follow it in `fluxions`. These policies include the names of functions, the names of arguments, and order of arguments. Why `numpy`? It's a successful and high quality library that is in wide use. Almost all of the user base for a Python library that does automatic differentiation will also be using `numpy`. The more our library looks and acts like `numpy`, the easier it will be to use. It will also simplify implementing it with `numpy` as the back end.

Software Organization

[directory structure ??]

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 $\mathbb{R}^m$.

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`.

[This section needs more]

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`.