CS207 - Milestone 1

Introduction

We present `autodiff`, a modern, clean, fast implementation of automatic differentiation.

Differentiation is ubiquitous in sciences and engineering. But taking derivatives on a computer is a nontrivial problem. Enter: automatic differentiation (AD). AD is a set of techniques used to numerically evaluate the derivative of a function specified by a computer program.

Applications of AD include real-parameter optimization (through gradient-based methods), physical modeling (e.g. forces are derivatives of potentials, equations of motion are derivatives of Lagrangians and Hamiltonians, etc), probabilistic inference (e.g. Hamiltonian Monte Carlo), and machine learning (neural networks). Due to the computational cost of these tasks, it is necessary to implement a reliable and efficient differentiation utility. AD allows us to avoid the complexity of symbolic differentiation as well as the numerical instability of finite difference schemes. It is a true asset for better understanding scientific computing applications and predictive science.

Background

Broadly, there are three approaches to taking a derivative on a computer:

1. Symbolic differentiation
2. Numerical differentiation
3. Automatic differentiation

Symbolic differentiation computes a derivative directly from the underlying analytical expressions. This approach can compute the derivative exactly, but requires manually specifying the derivatives in the implementation, and can be quite slow and brittle. Numerical differentiation, while potentially fast, can suffer from accuracy and stability issues in certain problem spaces. Finite differences for instance can induce both truncation and round-off errors, even with an optimized step size.

Automatic differentiation combines the best of both worlds, finding exact derivatives while maintaining speedy computation. AD itself works by computing traces along a computational graph, and using the chain rule to iteratively compute derivatives. It breaks down the original function into child functions which are analytically computed and combined through simple arithmetic operations. The resulting individual derivatives computed are much simpler and faster.

In AD, the main mathematical concepts include chain rule, Jacobian matrix, linear algebra, dual numbers, Taylor Series Expansion, reverse and forward passes through the computational graph.

Here is a brief overview of the main mathematical concepts mentioned above:

- In calculus, chain rule allows us to take the derivative of composite functions. It is a key underlying concept of automatic differentiation. For a one-dimensional function vector $h(u, v)$ such that $u := u(t)$ and $v := v(t)$, we can take the derivative of h with respect to t as:

$$\frac{\partial h(u(t), v(t))}{\partial t} = \frac{\partial h}{\partial u}\frac{du}{dt} + \frac{\partial h}{\partial v}\frac{dv}{dt}$$

- When we wish to return the derivative of a multidimensional function with respect to multiple variables as a single entity, we need to compute the Jacobian matrix. For an m-dimensional function vector and a n-dimensional variable vector, we obtain an $m \times n$ $\boldsymbol{J}$ matrix where:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial \boldsymbol{f}}{\partial x_1} & \cdots & \frac{\partial \boldsymbol{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- Linear Algebra is about linear combinations, where we use arithmetic on columns of numbers called vectors and arrays of numbers called matrices.
- Dual numbers are expressions of the form $z = a + b\epsilon$, where a and b are real numbers, and ϵ is a symbol taken to satisfy $\varepsilon^2 = 0$. Through the Taylor Series Expansion of $f(z)$ where z is taken as before and expanding around $c = a + 0\epsilon$, we can show that:

$$f(z) = \sum_{k=0}^{\infty} \frac{f^{(k)}(c)}{k!}(z-c)^k = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(b\epsilon)^k = f(a) + f'(a)b\epsilon$$

Note that all higher order terms vanish due to $\epsilon^2 = 0$. Hence, passing the dual number $a + b\epsilon$ as an argument to a function f will result in another dual number of the form $f(a) + f'(a)b\epsilon$ where $f(a)$ is the function value and $f'(a)$ is the function derivative value. This is the motivation behind using dual numbers as our main data structure for our AD software package; dual numbers will be propagated forward through the computational graph, transmitting both function value and derivative value. The former is referred to as the primal trace while the latter is tangent trace.

- A computational graph is a directed graph where the nodes correspond to operations or variables. Variables can feed their value into operations, and operations can feed their output into other operations. This way, every node in the graph defines a function of the variables.
    - Forward pass is the procedure for evaluating the value of the mathematical expression represented by computational graphs. Doing forward pass means we are passing the value from variables in forward direction from the left (input) to the right where the output is. In forward mode, we compute function and derivative values simultaneously as we move right on the computational graph from inputs to outputs. We will also need to define the direction of forward pass that is of interest.
    - In the backward pass, our intention is to compute the gradients for each input with respect to the final output. In reverse mode, we move forward to compute function values before traversing back, computing derivatives at every node. We use reverse mode when the dimension of the input is larger than that of the output.

How to use `autodiff`

Here's an example of using the `autodiff` package:

```
from autodiff.forward import symbol, function as f

x, y = symbol('x y')
expression = f.sin(x) * f.tanh(y)

# x and y are hashable
p = {x: 1, y: 0}
result = expression.deriv({x: 5, y: 0}, seed=p)

print('The derivative is: ', result)
```

The syntax is inspired by SymPy. Here's a line-by-line guide to this snippet:

First, we define the set of symbols we would like to use in our AD problem:

```
x, y = symbol('x y')
```

The function `symbol()` takes as input a string of space-separated values. Each value becomes a new "symbol" that can be used in a math expression on which a derivative will eventually be computed. `symbol()` returns a tuple of symbols, which can be unpacked into variables.

Next, we create an expression object.

```
expression = f.sin(x) * f.tanh(y)
```

To evaluate the derivative for a particular `x` and `y`, we can call

```
p = {x: 1, y: 0}
result = expression.deriv({x: 5, y: 0}, seed=p)
```

If the seed parameter is not passed, the function `deriv()` will infer the correct shape of the derivative to return. For example, if `expression` is a scalar function with scalar output, `deriv()` will return a scalar value. If `expression` is a scalar function with vector input, `deriv()` will return a vector (the gradient). And if `expression` is a vector function with vector input, `deriv()` will return a matrix (or more precisely, a `numpy.ndarray`, which is the Jacobian).

See below for additional implementation details about these functions.

Software Organization

Here's how our project repo will be organised:

```
cs107-FinalProject/
├─ docs/
│  ├─ milestone1.md
│  ├─ milestone2.md
├─ LICENSE
├─ README.md
├─ setup.py             # or setup.cfg depending on later preference
├─ src/
│  ├─ __init__.py
│  ├─ forward/
│  │  ├─ __init__.py
│  │  ├─ symbol.py
│  │  ├─ function.py
│  ├─ backward/          # to be implemented at a later date
│  │  ├─ __init__.py
│  │  ├─ symbol.py
│  │  ├─ function.py
├─ tests/
│  ├─ run_coverage.sh
│  ├─ run_tests.sh
│  ├─ forward/
│  │  ├─ test_symbol.py
│  │  ├─ test_function.py
├─ .travis.yml
```

As described above, for the initial stage of the project, we will be developing functionalities for forward mode of automatic differentiation.

The most important module is the `symbol()` module, which will be the main module for the user to define and introduce independent variables that are of interests. Nevertheless, `symbol()` can also represent dependent variables, allowing storage of dependent variables' intermediate values and derivative values. Users will be able to assign values to the independent variables and return the value of primal trace and tangent trace after the function calculation.

The `function()` module will take an input of `symbol()` variable, process the function expression, and output the dependent variable as a `symbol()` variable. This `function()` module will include commonly used math functions. Overloading of more common mathematical operations happens in `symbol()` module.

To ensure the integrity of the project and streamline the testing process, we will follow the Continuous Integration process. `unittest` will be the main helper package to conduct unit tests on modules and functionality of our package. Further, `TravisCI` will be the main platform to receive frequent unit test results and `CodeCov` for code coverage report. Test suites will be under each sub package folder in the tests folder.

Finally, we will publish our package on PyPI for distribution. And for distribution testing, we will be using https://test.pypi.org/. `setuptools` will be the main tool to package the software and a `setup.py` file will be included in the root directory to import `setuptools` and specify the package requirements.

The python versions supported will be 3.6, 3.7 and 3.8.

Implementation

To implement automatic differentiation, the core data structure we use is the dual number (see above for additional details about how dual numbers can be used to compute AD) and ndarray for matrix calculation and manipulation.

Symbols and functions

In the code, dual numbers are implemented as instances of the `Symbol` class (or its subclasses). To make a symbol, a user can simply call `x = symbol('x')`. The string passed to `symbol()` represents the name of the variable. The resulting `Symbol` object is stored in the Python variable `x`. To make multiple symbols at once, a user can call `symbol()` with a string of space-separated names

```
x, y, z = symbol('x y z')
```

Dunder methods are implemented on `Symbol` to return other symbols from basic operations

```
expression = x + y   # x and y are Symbols
```

In this example, `expression` is also a `Symbol`. The results of these operations are guided by the way dual numbers can be manipulated.

To implement more advanced functions, we implement a wide array of elementary functions in the subpackage `autodiff.forward.function`. `function` contains a variety of Python methods, each with a common signature:

```
def f(input: Symbol) -> Symbol:
    # computes the resulting dual number
```

One example method in this subpackage is `sin()`. Upon importing `sin()` into the current namespace, a user can call `expression = sin(x)`, and obtain a dual number encapsulating this computation. Note, for each of these custom functions, we will necessarily implement a subclass of `Symbol` that implements its derivative correctly.

Under this scheme, specifying functions with vector inputs is as easy as specifying an expression with multiple symbols:

```
expression = x ** 2 + y * 2    # function of two variables
```

To specify a vector function, there's a special function defined in `function` that concatenates multiple `Symbol`s.

```
def concat(*symbols:) ->  Symbol:
    # returns a Symbol composed of multiple inner Symbols
```

So a vector function could look like

```
vector_function = f.concat(x**2 + y**2, x + y)
```

To figure out how many `Symbol`s may be nested within a single `Symbol`, you can call `len()` on a `Symbol.`

```
len(vector_function) == 2   # True
```

More generally, to print the number of inputs and outputs in a vector function `Symbol`, we can use the `shape` method

```
vector_function.shape == (2, 2)  # (two outputs, two inputs)
```

Evaluating the derivative

An instance of `Symbol` has a method `deriv()`. This method has the following signature:

```
def deriv(self, inputs: dict, seed: dict) -> Union[float | np.ndarray]:
    pass
```

When `deriv()` is called on a symbol, the derivative is computed on the expressions accumulated in the symbol, and a result is returned. If a `seed` parameter is supplied, the `deriv()` function will take into account the starting seed when computing the final result. `deriv()` also infers the correct shape of the derivative. For example, if `x` is a vector function with vector inputs, and no seed is passed, then `x.deriv({})` will return a

`np.ndarray` matrix (the Jacobian). Note, if no parameters are passed in the dict, the derivative is evaluated at 0.

How exactly this evaluation occurs is worth brief elaboration. Necessarily, computing this derivative must occur lazily. To do so, rather than hold explicit values within the dual number, we hold function instead that take as input the inputs of the `Symbol`, and return a concrete value when called. Hence, when a dual number / `Symbol` needs to be evaluated, evaluation will occur as simply a function call. So then, rather than have members set as explicit values:

```
# Symbol init function
def __init__(self, real: float, dual: float):   # not lazy-evaluated
    self.real = real
    self.dual = dual
```

We have them set as functions instead

```
# Symbol init function
def __init__(self, real: Function, dual: Function):   # lazy-evaluated
    # (note the actual implementation will differ, this is for demonstration only)
    self.real = real
    self.dual = dual

# Then when the time comes:
def deriv(inputs: dict, seed=None) -> Union[float | np.ndarray]:
    # some important code...
    realized_value = self.dual(inputs['variable_name'])
    # more important code...
```

Licensing

We decided to use the MIT license because it is a very open, permissive license with very few restrictions. We want our package to benefit as wide an audience as possible, whether commercial, for-profit, hobbyist, free software, or otherwise. By using the MIT License, we guarantee this code can help just about anyone who needs it without any legal obstructions.

Feedback

Milestone 1

From canvas:

```
Introduction[2/2]
Background[2/2]
How to use[3/3]
Software Organization[3/3]
Implementation[3/3]
```

```
License[2/2]

Sergey Litvinov , Oct 29 at 2:55pm
```

No specific feedback for improvements was given, so we've taken the opportunity instead to polish our existing work in preparation for future milestones.