Documentation

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`

Installation

Install the latest stable release with `pip`:

```
pip install AutoDiff-jnrw
```

If you would like the latest development version, you may also install the package directly from the GitHub repo. To do so:

1. Clone the source from GitHub.
2. Change the directory into the cloned folder and run the following command to install the package.

```
pip install .
```

3. To start using the package, the recommended way is to import as

```
import autodiff.forward as fw
import autodiff.backward as bw
import autodiff.vector as vc
```

Usage

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

```
import autodiff.forward as fw

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

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

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 = fw.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 = fw.sin(x) * fw.tanh(y)
```

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

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

If the seed parameter is not passed, the function `deriv()` will return a dictionary object mapping variables names to their partial derivatives.

See below for additional implementation details about these functions.

Testing

This project uses `pytest` for testing. To run the testing suite, from the top-level directory simply call

```
pytest
```

To generate a browsable code coverage report, do

```
coverage run -m pytest
coverage html
```

Software Organization

Here's how our project repo will be organized:

```
cs107-FinalProject/
├─ docs/
│  ├─ documentation.md
│  ├─ milestone1.md
│  ├─ milestone2.md
├─ LICENSE
├─ README.md
├─ setup.cfg
├─ requirements.txt
├─ src/
│  ├─ __init__.py
│  ├─ autodiff/
│  │  ├─ __init__.py
│  │  ├─ forward/
│  │  │  ├─ __init__.py
│  │  │  ├─ sym.py
│  │  │  ├─ function.py
│  │  ├─ backward/
│  │  │  ├─ __init__.py
│  │  │  ├─ sym.py
│  │  │  ├─ function.py
│  │  ├─ vector/
│  │  │  ├─ __init__.py
│  │  │  ├─ sym.py
│  │  │  ├─ function.py
├─ tests/
│  ├─ forward/
│  │  ├─ test_symbol_fw.py
│  │  ├─ test_function_fw.py
│  │  ├─ test_newton_fw.py
│  ├─ backward/
│  │  ├─ test_symbol_bw.py
│  │  ├─ test_function_bw.py
│  ├─ vector/
│  │  ├─ test_sym_vc.py
│  │  ├─ test_function_vc.py
├─ demo/
│  ├─ NN.py
├─ .circleci.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.

Further to perform calculations in vectors, `vector` module will be main module to interact with. It can create symbol vectors `SymVec` for vector operations and also enable concatenating individual symbols in to a such vector. It also supports application of mathematical functions to the vectors.

To ensure the integrity of the project and streamline the testing process, we will follow the Continuous Integration process. `Pytest` will be the main helper package to conduct 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 Details

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).

Symbols and functions

In the code, dual numbers are used by instances of the `Symbol` class to store and compute derivatives. 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 = fw.symbol('x y z')
```

Symbols with the same name are not supported, and will err if the user attempts to create them in the same call.

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`. Other operators overloaded for `Symbol` include:

- `+` addition
- `-` subtraction, unary negation
- `*` multiplication

- `/` division
- `**` power

With the exception of `**`, all operations above work with any combination of `Symbol` and numeric operators. `**` will work with one symbolic and one numeric operator in any order (`np.e ** x` and `x ** np.e` are equally valid), or with two symbolic operators.

To use 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 fn(a: Symbol, b: Symbol) -> Symbol:
    # computes the resulting trace
```

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.

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
```

Symbol internals

Under the hood, `Symbol` is a node in a computational graph, tracking up to two parent nodes together with an operation to produce the current value. When it comes time to evaluate the derivative, a `DualNumber` object is percolated through the graph, recording the primal and dual traces of the computation. When a function is applied to one or two `Symbol`s, a new `Symbol` is produced with the original nodes as parents, along with an operator (in practice, just a Python function). When an evaluation call is made to a `Symbol`, the `Symbol` traces through its parents to the root, then percolates a `DualNumber` object back to the start. See below for more details about the derivative evaluation.

Evaluating a `Symbol`

Every `Symbol` as two methods for evaluation: `eval()` and `deriv()` . `eval()` as the following signature:

```
def eval(self, inputs: dict) -> float
```

`inputs` is a `dict` mapping variables names to the values they should assume in a computation. So for example, if we have the symbol `f = x + y`, where `x` and `y` are also symbols, we can evaluate `f` by calling `f.eval({'x': 1, 'y': 2})`.

The `deriv()` method has the following signature:

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

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, scaling the resulting outputs along each variable by the value specified in `seed`. In this case, `deriv()` will return a `float` representing the directional derivative. If the `seed` dictionary is missing a particular variable name, it is assumed that the weight assigned to that variable should be 0.

If `seed` is not specified, `deriv()` will return a `dict` object that maps variable names to their derivatives. So if `f = x + y`, then `f.deriv({'x': 1, 'y': 2})` will return `{'x': 1, 'y': 1}`.

Currently, vector functions are unimplemented. When they are, rather than return a single `float` or `dict` object, `deriv()` returns a `list` of these objects, corresponding to their order in the vector function.

`SymVec` and Functions

`SymVec` is an ensemble of multiple `Symbol` that supports vector operations and vector derivatives.

There are two ways to initiate a `SymVec` object.

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

```
import autodiff.vector as vc


def concat(symbols: List[Symbol]) ->  SymVec:
    # returns a Symbol composed of multiple inner Symbols
```

So a vector function could look like

```
a, b = fw.symbols('a b c')
v1 = vc.concat([a ** 2 + b ** 2, a + b, a * b])
```

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

```
len(v1.names) == 2   # True
```

More generally, to print the number of outputs in a vector function `SymVec`, we can use the `shape` method or `len()` on the vector.

```
# three outputs
v1.shape == 3
len(v1) == 3
```

Further, for a quick initiation, user can use `vec_gen` generator that is provided by the module.

```
g = vc.vec_gen()
# to initiate a Symbol Vector of dimension 3
v1 = g.generate(dim=3)

assert v1.names == ['dum00000', 'dum00001', 'dum00002']
```

The naming for auto generated vector symbol will follow a convention of 'dum' + counter.

Evaluating a `SymVec`

Every `SymVec` as four methods for evaluation: `eval()` and `deriv()`, `quickeval()` and `quickderiv()` . `eval()` and `quickeval()` are as the following signatures:

```
def eval(self, inputs: dict) -> np.ndarray
# quickeval will pass in the mapping of array value to the sorted order of symbol names
def quickeval(self, inputs: np.ndarray) -> np.ndarray
```

In eval method, `inputs` is a `dict` mapping variables names to the values they should assume in a computation. And in `quickeval`, the np array represent the sorted order of symbol names and their values. So for example, v1 = vc.concat([a,b,c]), and we pass in an input of np.array([1,2,3]). The value of the array will be mapped to the sorted order of symbols, that is the same as passing in a dict for eval method of {'a':1, 'b':2, 'c':3}

The `deriv()` and `quickderiv()` methods have the following signatures:

```
def deriv(self, inputs: dict) -> np.ndarray()
def quickderiv(self, inputs: np.ndarray) -> np.ndarray
```

When `deriv()` is called on a symbol, the derivative is computed on the expressions accumulated in the symbol, and a result is returned. And `quickderiv` is following a similar passing in manner as `quickeval`. The biggest difference is that, the output is now is shown as a Jacobian matrix. The order of the array is in correspondence as the sorted order of related symbols.

Extension features

As an extension on top of the project requirements, we implement reverse mode automatic differentiation.

For ease-of-use, the API for using reverse mode is identical to that of forward mode. If a user desires to switch their existing code from forward to reverse mode, they need only change their import to the relevant package

```
import autodiff.backward as bw

# Now the code below uses backward-mode to compute the derivatives
x, y = bw.symbol('x y')
expression = bw.sin(x) * bw.tanh(y)

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

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

All functions present in the forward mode are preserved for the `backward` package. As the API's are identical, any code written using forward mode can be easily switched to a reverse mode implementation.

Implementation details

Under the hood, the reverse mode implementation operates very differently than the forward mode implementation. A computation graph is still maintained, but rather than percolate a dual number through the graph, we implement two passes across the pass to implement the reverse mode computations.

In the forward pass, each node is evaluated at the initial point specified by the user, and the partial derivatives relative to the children nodes are computed and stored. Also during the forward pass, each node is set with a counter indicating the number of parents it has. This counter is then used during the backward pass to track whether all dependent nodes have accumulated their full derivative prior to using this value. Computing just the forward pass is sufficient to evaluate the function for a particular input. But because of the side effects that necessarily accumulate in each node during this pass, we also expose a `symbol.zero()` method on each `backward.Symbol` instance. Calling this method with erase all gradients and reference counts in stored in each node.

To compute the full derivative, a user will need to use both the forward and backward pass. In the backward pass, the derivatives tracked in each node are accumulated to compute the overall derivative of the function with respect to each of the root input nodes.

The overall API of the reverse mode implementation is identical to the forward mode implementation, so the end user need never (nor should) call the `_forward()` and `_backward()` methods directly. Rather, as in the forward mode implementation, the user should call `eval()` and `deriv()` to obtain the function evaluation and derivatives respectively for a particular function.

Demo: Neural Networks

In addition to implementing the reverse mode, we also provide a short demo using the `autodiff` package for a real-world application: neural networks. This demo implements two neural network architecture

1. 1 hidden layer, linear activation
2. 2 hidden layers, ReLU activation

The neural network models are trained on a simplified MNIST dataset. The full demo is available at in the Jupyter notebook `demo/NN.py` and will require `NumPy` and `sklearn` to run.

Broader Impact and Inclusivity Statement

Broader Impact

Automatic differentiation has many applications in a variety of disciplines of engineering and science. It is a key component of many optimization algorithms that are used in dynamics, physics as well as computer science and Machine Learning. Its impact therefore goes beyond simple derivative cases and has far-reaching implications in these scientific domains.

As a result, it is important to acknowledge that this package is merely a tool that was developed purely for academic purposes to help us better understand the system development process. We understand that this package could also be misused and could have negative implications for individual privacy and well-being. Examples of misuse include Machine Learning algorithms that could invade individual privacy for political or financial profit.

Despite the above, we believe our package should still be distributed given the importance and widespread use of automatic differentiation. In order to avoid potential ethical violations, we suggest the implementation of a screening process that would evaluate the safety of using the package for a specific application.

Software Inclusivity

We believe in the importance of inclusivity in software development. We strived to make it as transparent and simple as possible for other developers to use and contribute to our code base. This package welcomes and encourages usage from disparate fields and various cultural, social and economic backgrounds. We acknowledge however that documentation has only been implemented in English and therefore requires basic understanding of that language. Any effort to make it more universally understandable is welcome. During the software implementation, we have worked closely as a group, respecting each contributor as well as carefully reviewing Pull Requests before merging. We will make sure to follow the same etiquette with outside contributors that may want to improve the package. We hope to have at least one member of the group review each PR. We also invite outside developers to be respectful with one another and resolve any conflict that may arise as professionally as possible. Please understand that we are all full-time students and as such, approval and review of pull requests may be delayed.

Future

As an extension to our feed-forward neural network application, we could also implement new integration tests pertaining to different areas of applications such as Hamiltonian Monte Carlo sampling, numerical optimization or variational inference. These tests would help demonstrate both the robustness of our implementation as well as the ease of use of our package.

Furthermore, we would have liked to add a visualization functionality to our library that would enable the user to see the resulting computation graph and/or the evaluation trace table illustrating the underlying computations of Automatic Differentiation. This would also be a great academic tool to better understand the algorithm.

Along the same lines, we would love to see a graphical user interface of our package implemented. The user could simply enter the function and the variables required for derivation and our library would work in the backend to output the results along with a computation graph or an evaluation trace table. The GUI would make it easier for users less familiar with programming to run our package.