AC 207 Final Project: Milestone 1

Group 22: Neil Sehgal, Lotus Xia, Andrew Zhang, Diwei Zhang

October 2021

1 Introduction

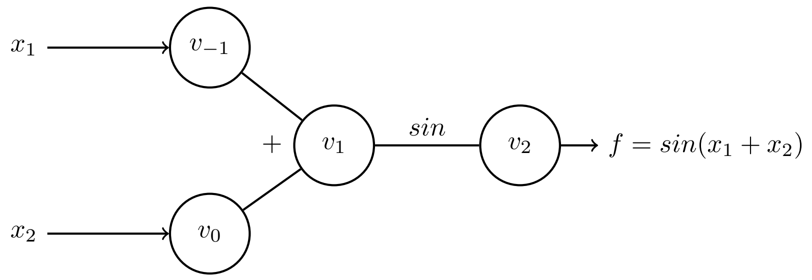
Differentiation is a central operation in science and engineering, used in various settings from financial markets to physics labs. For example, to optimize a function we must find the minima or maxima by using differentiation tests. There are multiple computational mechanisms to find derivatives. For example, numerical approximations like the Finite-Differences method involve utilizing the definition of a derivative and plugging in small values of h to evaluate the function. However, this method suffers from poor accuracy and stability issues due to floating point errors. Symbolic computation is another method, more accurate than numerical approximations, but often not applicable due to its unweildly and complex overhead. Here, we focus on automatic differentiation (AD), which overcomes the drawbacks of these two issues: it has less overhead than symbolic differentiation while calculating derivatives at machine precision. This makes it an integral part of machine learning algorithms, which demand computational efficiency as well as high precision.

2 Background

Automatic differentiation can be approached in two ways, known as *forward mode* and *reverse mode*. Both methods depend on the decomposition of a complex function into elementary operations. Then, we can take advantage of the chain rule to compute the derivatives of complex functions from trivially computed derivatives of these elementary functions.

2.1 Computational Graph

The first step of AD is to construct a computational graph which relates the elementary inputs to the final function. Each node represents an intermediate computation v_n, and edges connect nodes via elementary functions V_n. For example, given a multivariate scalar function $f(x_1, x_2) = sin(x_1 + x_2)$, the computational graph looks like the following:



Elementary functions include but are not limited to the following: algebraic operations, exponentials, trigonometrics, and their inverses. AD takes advantage of the fact that derivatives of elementary functions are trivial to compute, allowing the differentiation of complex functions via the chain rule.

2.2 Chain Rule

Suppose we want to calculate $\frac{\partial f}{\partial x_1}$. From the computational graph, we can rewrite f in terms of nested intermediate functions:

$$f(x_1, x_2) = V_2(V_1(V_{-1}(x_1), V_0(x_2)))$$

where V_n is the function that gives the value of node v_n. The chain rule states that

$$\frac{\partial f}{\partial x_1} = \frac{\partial V_2}{\partial V_1} \frac{\partial V_1}{\partial V_{-1}} \frac{\partial V_{-1}}{\partial x_1}$$

For a multivariate function $f(g_1(x), g_2(x), ..., g_n(x))$, the chain rule generalizes to:

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \left(\frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x} \right)$$

2.3 Forward Mode

Forward mode, which is the conceptually simpler method, involves computing the chain rule from inside to outside. For the above example $f(x_1, x_2) = sin(x_1 + x_2)$, the *evaluation trace* at the point $(1, 2)$ is generated using the computational graph as a guide:

trace	**elem**	**val**	**elem_der**	∇_{x_1}	∇_{x_2}
$v_{-1} = x_1$	x_1	1	$\dot{v_{-1}}$	1	0
$v_0 = x_2$	x_2	2	$\dot{v_0}$	0	1
v_1	$v_{-1} + v_0$	3	$\dot{v_{-1}} + \dot{v_0}$	1	1
$v_2 = f(x_1, x_2)$	$\sin(v_1)$	$\sin(3)$	$\cos(v_1)\dot{v_1}$	$\boxed{\cos(3)}$	$\boxed{\cos(3)}$

The evaluation trace essentially computes a value (primal trace) and derivative (tangent trace) for each node of the computational graph. The desired final results are $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$, shown in the last row of the trace using the gradient symbol ∇. Forward mode is advantageous when there are few inputs (m) and many outputs (n). The opposite is true for reverse mode, which motivates its use in deep learning applications with many inputs.

2.4 Gradient & Jacobian

The gradient of a multivariate scalar-valued function $f(g_1, g_2, ..., g_m)$ gives all of its partial derivatives in a vector of size $m \times 1$:

$$\nabla f = \nabla f_g = \begin{bmatrix} \frac{\partial f}{\partial g_1} \\ \frac{\partial f}{\partial g_2} \\ \vdots \\ \frac{\partial f}{\partial g_m} \end{bmatrix}$$

If the inputs g_i are themselves multivariate functions with inputs $x \in \mathbb{R}^m$, the chain rule gives

$$\nabla_x f = \sum_{i=1}^{n} \left(\frac{\partial f}{\partial g_i} \nabla g_i(x) \right)$$

Finally, we reach the most general form of the gradient: for multivariate vector-valued functions with m inputs and n outputs, the derivative of outputs with respect to inputs can be represented in a *Jacobian matrix* of dimensions $n \times m$:

$$f(x_1, x_2, ..., x_m) = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$J_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

2.5 Dual Numbers

Dual numbers offer a convenient way to store and compute the primal and tangent traces in forward mode AD. A dual number has the form

$$a + b\epsilon,$$

where $a, b \in \mathbb{R}$ and $\epsilon^2 = 0$. Combined with Taylor series expansion of a function f centered at a, dual numbers give rise to the property

$$f(a + b\epsilon) = f(a) + f'(a)b\epsilon$$

because all terms with powers of ϵ higher than 2 are equal to zero. Thus, the Taylor series approximation of a function about a dual number gives both its value and derivative, in a single calculation. In forward mode AD it is convenient to represent both the primal and tangent trace of each node as a dual number, where the real part corresponds to the primal and the dual part corresponds to the tangent. As shown below, functions (operations) on these nodes preserve the relationship between the real and dual parts:

$$g\big(f(a + \epsilon)\big) = g\big(f(a) + f'(a)\epsilon\big) = g\big(f(a)\big) + g'\big(f(a)\big)f'(a)\epsilon$$

Note that the coefficient of ϵ is simply the derivative of the real part $g(f(a))$, as given by the chain rule.

3 How to Use `DaLand`

3.1 Installation

The users will pull `DaLand` from PyPI using command line

```
pip install DaLand
```

Then import `DaLand` and other dependencies (e.g. `numpy`) in python script

```python
import DaLand as dl
import numpy as np
```

3.2 Methods

Users can use our package to deal with 3 different scenarios: 1 input to 1 output; multiple inputs to 1 output; multiple inputs to multiple outputs.

3.2.1 1 input to 1 output

Users may specify a single function with one input variable. See the example below with $f(x) = \exp(2x)$ with $x = 2$.

```
x = dl.Variable(2) # specifies the input variable
f = dl.exp(2 * x) # specifies functions involving the variable
```

We make two observations at this stage: First, arithmetic operations (exp in our example) need to be implemented within `DaLand`. Second, `f` is of type `Variable`.

To retrieve the value and derivative, call

```
val = f.get_value() # exp(4), float
der = f.grad() # 2exp(4), float
```

3.2.2 m inputs to 1 output

Users may also specify a single function with many input variables. See the example below with $f(x, y) = \exp(x + y)$, with $x = 1, y = 1$.

```
x = dl.Variable(1, label='x') # specifies input variable x
y = dl.Variable(1, label='y') # specifies input variable y
f = dl.exp(x + 2 * y) # specifies functions involving the variables

val = f.get_value() # exp(3), float
dx = f.grad(var = 'x') # exp(3), float
dy = f.grad(var = 'y') # 2exp(3), float
grad = f.grad() # {x: exp(3), y: 2exp(3)}, dict
```

When there are multiple input variables, users can specify the name of the variable when initializing the instance. The name usually matches the variable name on the left hand side. To get the derivative with respect to x, one simply specify `var = 'x'` within the function call. When no input variable is specified, `f.grad()` gives the gradient with respect to all input variables inside a dictionary.

3.2.3 m inputs to n outputs

We recognize that users may sometimes want to compute Jacobians. Consider the following case, $x = [x_1\ x_2]^T, y = [y_1\ y_2]^T, f(x, y) = [x_1 + 2y_1\ x_2 + 3y_2]^T$.

```
x1, x2, y1, y2 = 1, 2, 3, 4
x = dl.Variable([x1, x2], label='x') # specifies input variable x
y = dl.Variable([y1, y2], label='y') # specifies input variable y

f = x + dl.matmul([2,3], y) # specifies functions involving the variables
# f = [x1 + 2 y1, x2 + 3 y2]

val = f.get_value() #[[7],[14]]
grad = f.grad() # retrieve Jacobians for x and y
# {
#  "x": [[1,0], [0,1]],
#  "y": [[2,0], [0,3]]
# }
grad = f.grad(var = 'x') # [[1,0], [0,1]], np.array, Jacobians w.r.t x
grad = f.grad(var = 'y') # [[2,0], [0,3]], np.array, Jacobians w.r.t y
```

In this case, `f.grad()` gives the jacobian matrices with respect to x and y inside a dictionary. For instance, the (i, j)-th entry of the jacobian is $\frac{\partial f_i}{\partial x_j}$.

The current plan is to implement the forward mode auto-differentiation only. However, as we extend the functionality to include reversed mode auto-differentiation, we will allow users to choose, at the function definition stage, which auto-differentiation method to use.

```
x = dl.Variable(1, label='x') # specifies input variable x
y = dl.Variable(1, label='y') # specifies input variable y
f = dl.Variable(dl.exp(x + 2 * y), ad_mode = 'reverse') # specifies differentiation method
g = dl.Variable(dl.exp(x + 2 * y), ad_mode = 'forward') # specifies differentiation method
assert f == g
```

4 Software Organization

4.1 Directory Structure

We envision our directory structure will look something like the following:

```
cs107-FinalProject
 |__docs/
 |   |__documentation
 |   |__milestone1.pdf
 |   |__milestone2.pdf
 |__License
 |__README
 |__pyproject.toml
 |__setup.cfg
 |__src/
 |   |__ __init__.py
 |   |__ __main__.py
 |   |__utils.py
 |   |__daland.py
 |__tests/
     |__run_test.sh
     |__run_coverage.sh
     |__test_daland.py
     |__.travis.yml
     |__.codecov.yml
```

4.2 Modules

Three external modules that we will rely on in our implementation will be NumPy, pytest, and coverage.

- NumPy offers support for large, multi-dimensional arrays as well a wide range of optimized mathematical functions and operators for these arrays. Numerical calculation of function values in DaLand will rely extensively on this module.
- pytest is a feature rich testing framework in python that makes it easy to write unit tests that scale for full libraries. We will be using Travis CI to automatically run these tests.
- coverage is a common library used to compute code coverage during test execution. We will use CodeCov to display the results of coverage after TravisCI runs our tests.

4.3 Test Suite

The test suite will live inside the tests directory. It will run with pytest and will be automated with the principles of continous integration with TravisCI. On every push we make to GitHub, TravisCI will automatically run our tests. We will also make use of the coverage library to compute code coverage by lines and use CodeCov for display purposes.

4.4 Packaging and Distribution

We will distribute the package through the Python Package Index (PyPi). Users will be able to install our package just by doing:

```
pip install DaLand
```

5 Implementation

5.1 Data Structure

The core data structures used in our implementation are `list`, `numpy.array`, and `dict`. See the "m inputs to n outputs" example from above as an illustration. The input vectors are in `list`. The Jacobian with respect to each vector-form input is in `numpy.array`. To easily match the Jacobian to the corresponding input vector, we use a `dict`, where the key is the name of the corresponding input variable, and the value is a `numpy.array` of the Jacobian.

5.2 Classes and Methods

The baseline implementation currently includes a class `Variable`, in which we re-define dunder methods and other arithmetic operations. See below for an rough sketch of the class and method definition.

```
class Variable(object):
    def __init__(self, val, label=None, der=None, ad_mode='forward'):

        if label is not None:
          self.label = label
        else:
          self.label = 'v0'
        self.val = val

        if der is not None:
          self.der = der
        else:
          self.der = {label: [[1]]}

        self.ad_mode=ad_mode

    def grad(var=None):
        pass # output derivative/partial dev/Jacobian
    def get_value():
        pass # output function value
    def __add__(self, other):
        pass
    def __str__(self):
        pass
    def __radd__(self, other):
        pass
    def __sub__(self, other):
        pass
    def __rsub__(self, other):
        pass
    def __mul__(self, other):
        pass
    def __rmul__(self, other):
        pass
    def __truediv__(self, other):
        pass
    def __rtruediv__(self, other):
        pass
    def __neg__(self, other):
        pass
    def __pow__(self, other):
        pass
    def __rpow__(self, other):
        pass
    def __ne__(self, other):
        pass
    def __lt__(self, other):
        pass
    def __le__(self, other):
        pass
    def __gt__(self, other):
        pass
    def __ge__(self, other):
        pass
```

On a high level, each instance of this class corresponds to a node in the computation graph in the background section. We keep track of the primal trace (`self.val`) and the tangent trace (`self.der`) as attributes for all instances. When users call `.get_value()` or `.grad()` to retrieve function value and derivative, respectively, we simply return (formatted version of) `self.val` and `self.der` that have been calculated along the way.

The specific implementation details might change as we think more about the problem. The current plan is to keep `self.val` as an 1-D `numpy.array` and `self.der` as a dictionary, where the keys are the input variable names, and the values are 2-D `numpy.array`, even when the inputs and outputs are one-dimensional. For instance,

- For $f(x) = 2x$ at $x = 1$, we have `self.val = [2]`, `self.der =` {`'x': [[2]]`}.
- For $f(x, y) = x + y$ at $x = 1, y = 2$, we have `self.val = [3]`, `self.der =` {`'x': [[1]], 'x': [[1]]`}.
- For $f(x, y) = [x + y, x - y]$ at $x = 1, y = 2$, we have `self.val = [3, -1]`, `self.der =` {`'x': [[1], [1]], 'x': [[1], [-1]]`}.

We also implement elementary functions like `sin`, `sqrt`, `log`, and `exp` that will return `Variable` objects

```
def matmul(mat1, mat2):
    pass
def sin(x):
    pass
def cos(x):
    pass
def tan(x):
    pass
def exp(x):
    pass
```

```python
def log(x):
    pass
def ln(x):
    pass
def logistic(x):
    pass
```

A rough sketch of the `sin` function would look like the following:

```python
def sin(x):
    # x: a Variable object
    val = np.sin(x.val)
    der = np.cos(x.val) * x.der
    return Variable(val = val, der = der)
```

More arithematic operations and functions to be included if time allows and as we see fit.

6 Licensing

We will use the the MIT License. Future developers are welcome to extend our library and distribute closed source versions. Because we are using NumPy, a copyright licensed library, we are unable to use a license like GNU GPLv3 (as it requires all libraries used by the library to be copyleft).