1 Introduction

Automatic differentiation (AD or autodiff throughout) is a set of techniques for numerically evaluating a gradient. It achieves this by breaking the function-to-be-differentiated into a set of elementary operations and using the chain rule. We can contrast this with two other flavours of differentiation: symbolic and finite-differences. Symbolic differentiation aims to calculate a symbolic mathematical expression for the derivative of a particular function, while finite-differences numerically approximates a derivative at a particular point. Compared to the latter of these, AD achieves machine precision accuracy without sacrificing computational complexity and does not slow down as much when input dimensionality is high [1]. Compared to symbolic differentation, AD is not limited to closed-form functions, and can handle loops, recursion and control flow.

Not only is autodiff powerful compared to other differentiation techniques, but it is already used widely. Gradient-based optimization, using the negative gradient of a function to step towards its local minimum, is pivotal to deep learning implementations. These have come to dominate many machine learning domains including computer vision and natural language processing. Implementations of computational methods for probabilistic inference, such as Hamiltonian Monte Carlo and Variational Inference, have also come to rely on automatic differentiation for the computation of necessary gradients.

2 Background

We can consider any function that we may want the derivative of as being made up of elementary operations applied sequentially, such as arithmetic operations, switching sign, exponential, logarithmic, and trigonometric operators. We can easily compute the derivatives of such operations. Combining these derivatives through the chain rule gives the foundations of automatic differentiation.

2.1 The Chain Rule

In its simplest form, for real-valued functions of one variable, with $y = f(u)$ and $u = g(x)$ then the chain rule states

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Generalizing to an input vector x of dimensionality m and a scalar function f action on a vector of n other functions $y_i(x)$, we also have a more general form of the chain rule:

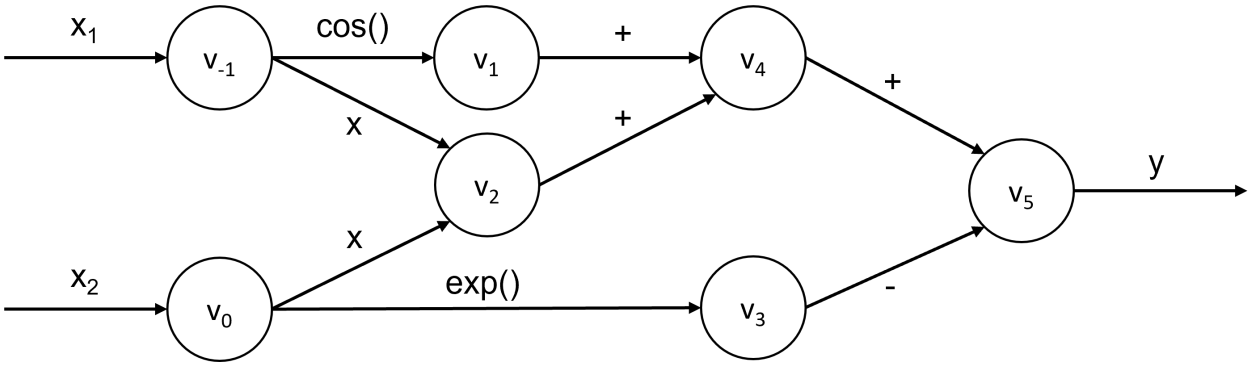
$$\nabla_x f = \sum_{i=1}^{n} \frac{\partial f}{\partial y_i} \nabla y_i(x)$$

2.2 Example

This example is motivated by [1]. Consider the function $y = f(x_1, x_2) = \cos(x_1) + x_1 x_2 - \exp(x_2)$ where we want to find $\frac{\partial y}{\partial x_1}$ at $(x_1, x_2) = (2, 4)$. We can decompose this problem into the steps in the table below.

Elementary operations	Derivative w.r.t x_1
$v_{-1} = x_1$	$\dot{v_{-1}} = 1$
$v_0 = x_2$	$\dot{v_0} = 0$
$v_1 = \cos(v_{-1})$	$\dot{v_1} = -\sin(v_{-1})\dot{v_{-1}}$
$v_2 = v_{-1}v_0$	$\dot{v_2} = \dot{v_{-1}}v_0 + v_{-1}\dot{v_0}$
$v_3 = \exp(v_0)$	$\dot{v_3} = \dot{v_0}\exp(v_0)$
$v_4 = v_1 + v_2$	$\dot{v_4} = \dot{v_1} + \dot{v_2}$
$v_5 = v_4 - v_3$	$\dot{v_5} = \dot{v_4} - \dot{v_3}$
$y = v_5$	$\dot{y} = \dot{v_5}$

The function can equivalently be displayed as a computational graph, where the nodes are intermediate variables and edges represent elementary operators:



Automatic differentiation can operate in two modes:

- Forward mode
  - The principal mode this package will focus on
  - Follow the chain rule from "inside" to "outside", i.e. in this example starting with $\frac{\partial v_{-1}}{\partial x_1} = 1$ and ending with $\frac{\partial v_5}{\partial x_1} = \frac{\partial y}{\partial x_1}$
  - Goes in the same direction as the evaluation of the function itself
  - Starts from the right-hand end of the chain rule expression above
  - Preferred when output space is much bigger than input space
- Reverse mode
  - Follow the chain rule from "outside" to "inside"
  - Goes in the reverse order as the evaluation of the function
  - Starts from the left-hand side of the chain rule expression above
  - Preferred when input space is much bigger than output space

The first column in the table is called the **Primal trace** and the second column is the **Forward tangent trace**. We can see how evaluating the intermediate steps (primals) as well as their derivatives (tangents) at the required point $(x_1, x_2) = (2, 4)$ starting from v_{-1} and $\dot{v_{-1}}$ eventually gives the required derivative evaluated at the correct point.

We can equivalently define a directional derivative operator $D_p y_i = \sum_{j=1}^{m} \frac{\partial y_i}{\partial x_j} p_j$, were we are projecting the gradient vector in the direction of a vector $\mathbf{p}$ with elements p_i. By selecting a p vector appropriate for the derivative we want, and letting the y_i be the intermediate variables in the trace, we achieve the same thing, except strictly now we are computing $\mathbf{J} \cdot \mathbf{p}$ were $\mathbf{J}$ is the Jacobian. It takes m passes to compute the full Jacobian.

2.3 Dual numbers

Forward mode autodiff can be achieved through as representing a function using dual numbers $v + \dot{v}\epsilon$ were $\epsilon \neq 0$ and $\epsilon^2 = 0$. The primal trace is carried in hte real part, and the tangent trace in the dual part.

This is a useful format because addition and multiplication of these dual numbers results in dual parts that respect differentiation rules w.r.t their real parts, e.g. $(v + \dot{v}\epsilon)(u + \dot{u}\epsilon) = (vu) + (v\dot{u} + \dot{v}u)\epsilon$. Most significantly, this formulation respects the chain rule, in that if we have $z_j = v_j + D_p v_j \epsilon$ then applying a function f gives $f(z_j) = f(v_j) + f'(v_j) D_p v_j$. So defining our dual number in this way we can progress through the intermediate steps of AD and the real and dual parts will give us the function and derivative evaluation naturally.

More detail can be found here.

3 How to use

Users will be able to access specific individual objects to use for AD but will also have access to a top-level class to perform end-to-end AD.

3.1 Automatic differentiation example

```
import autodiff30.autodiff30 as ad

@ad.adfunction(ndim = 2)
def foo(x):
    """
    x is a list-like object of size ndim
    foo is a vector function that returns an array of outputs
    """
    return_list = [ad.sin(x[0]), ad.cos(x[0]*x[1])]
    return return_list

x = [1,1]
foo_at_x = foo(x)
gradient_at_x = foo.grad(x)
```

3.2 Optimization example

```
import autodiff30.autodiff30 as ad
import autodiff30.optimize as ad_opt
```

```
@ad.adfunction(ndim = 2)
def foo(x):
    """
    x is a list-like object of size ndim
    foo is a vector function that returns an array of outputs
    """
    return_list = [ad.sin(x[0]), ad.cos(x[0]*x[1])]
    return return_list

initial_guess = [0,0]
min_f, min_inputs = ad_opt.SGD(foo, initial_guess)
# Optional parameters e.g. learning rate
```

4 Software Organization

- What will the directory structure look like?

```
autodiff30
|   x— LICENSE
|   x— README.md
|   x— pyproject.toml
|   x— setup.cfg
|   x— src
|   |   x— autodiff30
|   |   |   x— __init__.py
|   |   |   x— __main__.py
|   |   |   x— autodiff30
|   |   |   |   x— __init__.py
|   |   |   |   x— ad.py
|   |   |   |   x— dual.py
|   |   |   |   x— functions.py
|   |   |   x— optimize
|   |   |   |    x— __init__.py
|   |   |   |    x— optimization.py
|   x— tests
|   |   x— autodiff30
|   |   |   x— test_ad.py
|   |   |   x— test_dual.py
|   |   |   x— test_functions.py
|   |   x— optimize
|   |   |   x— test_optimization.py
|   x— examples
|       x— examples.py
|       ...
```

- What modules do you plan on including? What is their basic functionality?
  - ad: adfunction main class
  - dual: dual numbers objects and basic arithmetic
  - functions: math functions that can be used as building blocks
- Where will your test suite live?

  The tests in /tests will be executed through GitHub Actions.

- How will you distribute your package (e.g. PyPI with PEP517/518 or simply setuptools)?
  We will distribute it on the PyPI test server. Then, it can be installed with

```
python –m pip install –i
https://test.pypi.org/simple/autodiff30
```

- Other considerations?

5 Implementation

- What classes do we need? Which one will we implement first

  We need a class for dual number with all elementary operations implemented: mul, add, radd, rmul, div.... The two attributes of that class will be the real and the dual part. Since our implementation of AD entirely relies on dual numbers, we start by implementing the dual class. We will also have a decorator class that uses a dual number implementation to achieve automatic differentiation.

- What are the core data structures? How will you incorporate dual numbers?

  We will incorporate dual numbers through our class dual numbers. Our class will support an __init__ method that initializes the real and the dual parts of the dual number. The dual number class will be used in our implementation of AD in the adfunction class.

- What method and name attributes will your classes have?

  The dual class will support methods to compute elementary operations such as addition, subtraction, multiplication, divisions, along their swapped versions: __add__, __radd__, __mul__, __rmul__, __sub__, __rsub__, __pow__, __rpow__. The adfunction class will ave __call__ and grad methods.

- Will you need some graph class to resemble the computational graph in forward mode or maybe later for reverse mode? Note that in milestone 2 you propose an extension for your project, an example could be reverse mode.

  For our implementation of AD, we won't be needing a class to resemble the computational graph in forward mode. Since our proposed extension will not involve the reverse mode, we do not need to keep track of the graph in our implementation. We are going to build an optimization application for our package, which should not require a rap.

- Think about how your basic operator overloading template should look like. How will you deal with elementary functions like sin, sqrt, log, and exp (and many others)?

  Our implementation relies on dual numbers. However, most of the elementary functions such as sin, cos, sqrt... are coded on numpy, which does not support the type dual. We will therefore have to overload all NumPy functions so that they could take into input dual numbers. Practically, we will not be overloading NumPy functions, rather defining new functions that compute elementary operations (sin, cos, ln...) for dual number inputs.

- How do you want to handle cases for $f\colon \mathbb{R}^m \to \mathbb{R}$ or later $f\colon \mathbb{R}^m \to \mathbb{R}^n$? Would it make sense to design a high-level function object to model arbitrary functions f? You could think further and possibly plan for a grad() method or similar in a class that models ff, since computing the gradient (or Jacobian) is an operation that is often required.

  Our main class will have a grad method that will handle multivariate inputs and outputs.

- Do you want/need to depend on other libraries? (e.g. NumPy).

  We will import NumPy and rely on it for basic operations on supported type (e.g int, float...).

6 Licensing

The key library that will be used in our source code is NumPy, which is issued under a BSD 3-Clause "New" or "Revised" License, which is a generally permissive licence. We are implementing an automatic differentiation library. Many implementations of equivalent thins exist, likely with levels of optimization that ours does not have. AD is also already a part of popular packages like JAX. As such patents will not be something we need to consider. Ours is also not likely to be a huge program. As such we do not believe it necessary to have an especially strong licence or one that enforces copyleft. But given that the project originated within a Harvard class, we do not deem it appropriate that others should be able to advertise software that uses this code. Given that our project will mostly be using NumPy functions, we will use the same licence that they do.

7 Feedback

7.1 Milestone 1

- How to use: I would encourage you to add an example of how you would expect the user to use your optimization feature.
  - We have added pseudo code showing how to use the optimization element of the package in the How To Use section
- Implementation : I would expect to see a bit more on how you would expect the users to interact with your package. This will also help you to think in detail about the design of your functions, etc.
  - We believe this is covered in the How To Use section