CS207 Milestone 1

Group: 35 (Topcoder-Kitty-ML)
Members: Tamilyn Chen, Kar-Tong Tan, Mark Lock

1. Introduction

1.1 Overview

Derivatives play an integral role in computational science, ranging from its use in gradient descent, Newton's method, to finding the posteriors of Bayesian models. We discuss numerical differentiation, symbolic differentiation, how both demonstrate limitations, and automatic differentiation, the focus of our software. We acknowledge its effectiveness in both its accuracy and efficiency when evaluating derivatives. Lastly, we provide some real life applications in population dynamics, biology, and politics.

1.2 Motivation for Automatic Differentiation:

Because functions are often too complex to solve analytically, instead, we look to alternative methods that automatically calculate derivatives. There are three main ways to approach this issue: numerical differentiation from finding finite difference approximations, symbolic differentiation through expression manipulation, and automatic differentiation (AD or algorithmic differentiation). While numerical differentiation is easy to code, it is also subject to floating point errors; symbolic differentiation gives exact and accurate results, but is too computationally expensive. Thus, automatic differentiation proves to be the most effective method as it works to resolve both of these issues; AD is both exact/numerically stable and computationally efficient.

2. Background

2.1 What is AD?

Conceptually straightforward, automatic differentiation can be defined as a family of techniques that evaluate the derivative through the use of elementary arithmetic operations (ie. addition, subtraction, etc.), elementary arithmetic functions (ie. exp, log, sin), and the chain rule. AD is a recursive process that involves repeatedly taking the chain rule to the elementary operations at hand, and allows us to calculate the individual components of the gradient (a list of partial derivatives in terms of each of the inputs) evaluations, to produce results that are automatic and precise. There are two modes in AD: the **forward mode** and **backward mode**.

2.2 Forward Mode

The forward mode begins at the innermost portion of the function and repeatedly applies the chain rule while traversing out. There is a forward pass that creates the evaluation trace as well as finds the derivatives and the values of the partial derivatives at each step. Notably, the derivative at subsequent steps are calculated based on the derivatives calculated in preceding steps.

The forward mode can be simplified by utilizing another important component of automatic differentiation: dual numbers. Dual numbers are a type of number that uses ϵ and allows for simultaneously automatically differentiating a function while also evaluating the value of the function.

2.3 Reverse mode

In the backward mode, a forward pass creates the evaluation trace and indicates the partial derivatives at each step, but does not find the values of the partial derivatives. At the end of this process, the final node's derivative is evaluated by using an arbitrary seed. Then, the values of the partial derivatives that constitute the end node's derivative are found by performing a backward pass through the tree to get all the values. The reverse mode is significantly less costly than forward mode, particularly for functions with many input variables.

During both the forward and backward modes, all intermediate variables are evaluated, and their values are stored; these steps can be represented in a table, and further visualized in a computational graph. The graph (and table) essentially outlines this repeated chain rule process; it also serves as the basis of the logic behind our automatic differentiation software library.

2.4 Application of AD

AD can be applied to many branches of computational science, ranging in areas from population dynamics to political dynamical systems. Calculating the exact jacobian of a given function, AD provides important insight into the interactions between the variables in a dynamical system (ie. predator, prey populations), as well as a variable's long term behavior over time (ie. predator population in the absence of prey, competition food source model where a species drives another out).

3. How to Use ***KittyDiffy***

How do you envision that a user will interact with your package? What should they import? How can they instantiate AD objects?

3.1 Overview of AD library (KittyDiffy)

1. An AD class takes in a function when it is instantiated.
   Example: KittyDiffy("sin(x) +y")
   a. It will check whether it has trig, exp, power, basic functions - if not, returns error
2. The AD class will have two methods you can call on:
   a. Forward(*kwargs) each kwarg corresponds to a variable specified in the function - one can set the values for each variable this way. Example: KittyDiffy("sin(x) + y").Forward(x=1, y=1)
      i. evaluate() will evaluate the expression
      ii. trace() will output the evaluation trace
      iii. opcount() will output the operation count
3. No imports needed on user's part. This will be taken care of in _init_.py.

4. Software Organization

Discuss how you plan on organizing your software package.

4.1 What will the directory structure look like?

- tests/
  - helper_tests.py (for testing helper classes and methods)
  - forward_tests.py (for testing forward mode)
  - backward_tests.py (for testing backward mode)
  - interface_tests.py
- kittydiffy/
  - _init_.py (to load any package dependencies)
  - helper.py
    - code for instantiating class
    - for methods and or classes common to forward and backward modes - e.g. Jacobians, derivative rules
  - forward.py (module housing the forward methods for calculations)
  - backward.py (module housing the backward methods for calculations)
  - Interface.py (Interactive command line interface)
- biology_application/

    - scripts to demo use of kittydiffy on a biological application mentioned above
- README.md
- License.txt
- requirements.txt (specifying dependencies)

4.2 What modules do you plan on including? What is their basic functionality?

- We are planning on including two modules, one for forward mode and one for backward mode
    - Helper module: common classes and methods to both Forward and backward
    - Forward mode: evaluate a function at any number of inputs, but elementary functions will only take unary or binary inputs. Limited to only the following:
        - trig functions
        - exp functions
        - basic operators: multiply, add, subtract, division
        - Power functions (e.g. x^4 and 4^x)
        - All functions above can be used with vectors as we will utilize numpy arrays to do our transformations
    - Backward mode: will have same scope as forward mode
    - Command line interface
    - [Optional] it would be helpful to also have a function that returns the computation trace in an easy to read format for both of these modules.

4.3 Testing, packaging, and distribution

- Where will your test suite live? Will you use TravisCI? CodeCov?
    - We will use Travis CI and CodeCov
- How will you distribute your package (e.g. PyPI)?
    - We will use conda, specifically hosting on anaconda cloud (anaconda.org).
- How will you package your software? Will you use a framework? If so, which one and why? If not, why not?
    - We will use the conda framework since it’s currently the most widely recognized and easily accessible
- Other considerations?
    - Not at this moment but we will be sure to discuss these as we write our code

5. Implementation

Discuss how you plan on implementing the forward mode of automatic differentiation.

5.1 Overview

The high-level overview of how this works is that during the instantiation of the class, the user will provide the function as a string (e.g. “2*x+sin(y)”). Using regex, we will parse the string into the component parts to build the graph **(Figure 1)**. Note we will build in logic to take care of order of operations. We will also build in logic to take care of nested functions/operations such that the stopping condition in the recursion is to reach a single variable. Constants that have been modified by a function in the string (e.g. exp(2)) will be converted to a scalar constant for simplicity. Once the graph is built, we can then choose to differentiate according to forward or backward modes (if implemented).

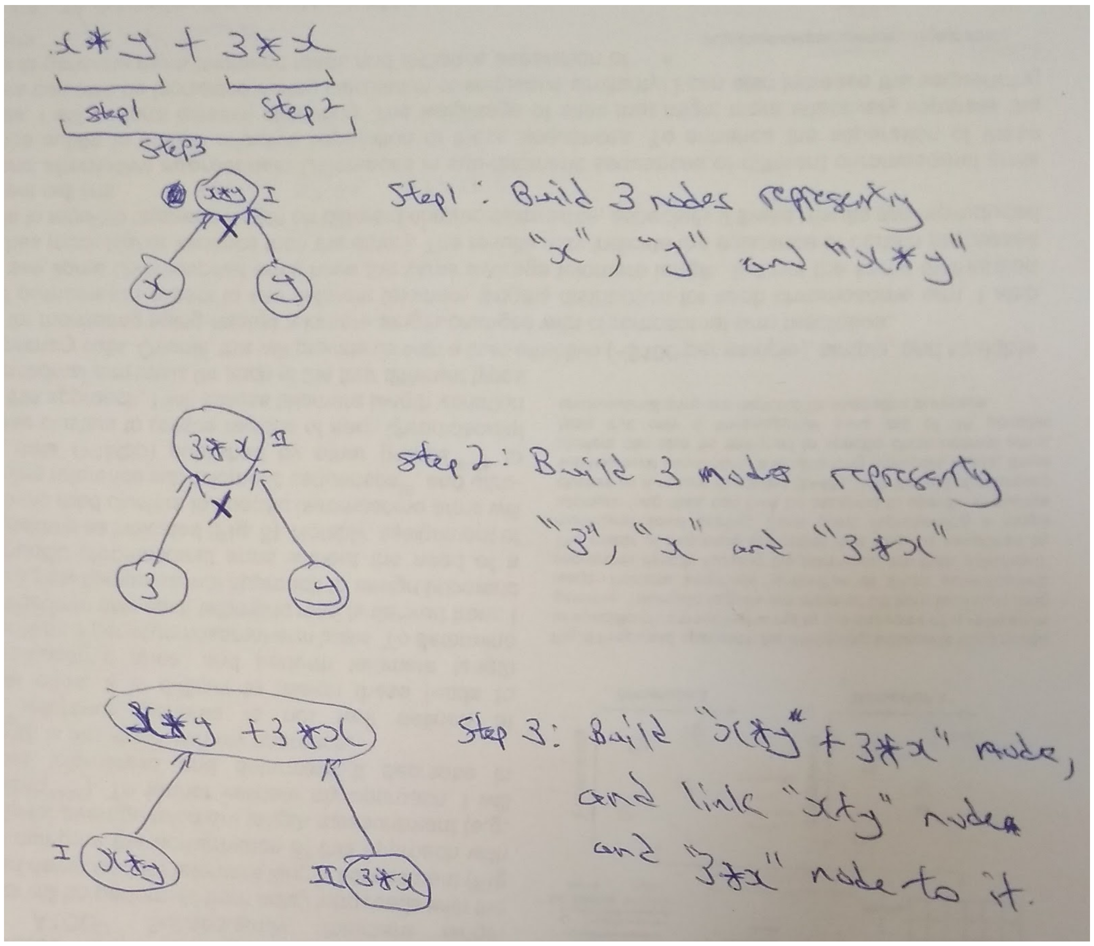
x*y + 3*x
step1
step2
step3
x*y
I
x
x
y
Step1: Build 3 nodes representing "x", "y" and "x*y"
3*x
II
x
3
y
step 2: Build 3 nodes representing "3", "x" and "3*x"
x*y +3*x
I
x*y
II
3*x
Step 3: Build "x*y + 3*x" node, and link "x*y" node and "3*x" node to it.

Figure 1 Example of how a full equation is converted to a pairwise input at each step of the evaluation. The nodes of the graph are built and linked to each other at each step of the process.

5.2 Data structures, classes and methods

- What are the core data structures?
    - Node class
    - The evaluation trace will be represented by a tree which will be a list of node classes
- What classes will you implement?
    - Graphical structure with nodes and edges (where each node is implemented by a class)
    - Each node will have the following attributes:
        - Edges list (List of list)
            - E.g. [[X1, in], [X2, out]]
            - X1, X2, etc. indicates the linking nodes
            - "In" and "out" indicates directionality (input vs. output node)
        - Elementary operation/function to be performed at this particular node (e.g. "+", "-", "*", "/", "sin", "exp", etc.)
- What method and name attributes will your classes have?
    - Name attributes
        - self.a (value of a for which we are evaluating the function at)
        - Self.value (value of the class at the value of a)
        - self.derivative (value of the derivative of the class at the value of a)
    - Method
        - Calculate function value at self.a
        - Calculate function derivative at self.a
        - __add__
        - __radd__
        - __mul__
        - __rmul__
        - __div__
        - __rdiv__

5.3 Dependencies

- What external dependencies will you rely on?
    - Python Math library (e.g. for math.pi, math.log, etc.)
    - Numpy
    - Re (regular expression) for parsing the elementary function which is provided by the user as a string.

5.4 More complex elementary functions

- How will you deal with elementary functions like `sin`, `sqrt`, `log`, and `exp` (and all the others)?
    - We will create a hard coded list of all elementary functions (at least the ones in our scope mentioned above) and their first order derivatives. That way, we can lookup in this table to provide the correct derivative transformation. For example, exp(x) will have a table entry that says the derivative function is 1/x.