AutoDiff Documentation (Milestone 2)

Nick Stern, Vincent Viego, Summer Yuan, Zach Wehrwein

Introduction

Calculus, according to the American mathematician Michael Spivak in his noted textbook, is fundamentally the study of "infinitesimal change." An infinitesimal change, according to Johann Bernoulli as Spivak quotes, is so tiny that "if a quantity is increased or decreased by an infinitesimal, then that quantity is neither increased nor decreased." The study of these infinitesimal changes is the study of relationships of change, not the computation of change itself. The derivative is canonically found as function of a limit of a point as it approaches 0 -- we care about knowing the relationship of change, not the computation of change itself.

One incredibly important application of the derivative is varieties of optimization problems. Machines are able to traverse gradients iteratively through calculations of derivatives. However, in machine learning applications, it is possible to have millions of parameters for a given neural net and this would imply a combinatorially onerous number of derivatives to compute analytically. A numerical Newton's method approach (iteratively calculating through guesses of a limit) is likewise not a wise alternative because even for "very small" h, the end result can be orders of magnitude off in error relative to machine precision.

So, one might think that a career in ML thus requires an extensive calculus background, but, Ryan P Adams, formerly of Twitter (and Harvard IACS), now of Princeton CS, describes automatic differentiation as "getting rid of the math that gets in the way of solving a [ML] problem." What we ultimately care about is tuning the hyperparameters of a machine learning algorithm, so if we can get a machine to do this for us, that is ultimately what we care about. What is implemented in this package is automatic differentiation which allows us to calculate derivatives of complex functions to machine precision 'without the math getting in the way.'

Background

The most important calculus derivative rule for automatic differentiation is the multivariate chain rule.

The basic chain rule states that the derivative of a composition of functions is:

$$(f \circ g)' = (f' \circ g) \cdot g'$$

That is, the derivative is a function of the incremental change in the outer function applied to the inner function, multiplied by the change in the inner function.

In the multivariate case, we can apply the chain rule as well as the rule of total differentiation. For instance, if we have a simple equation:

$$y = u \cdot v$$

Then,

$$y = f(u, v)$$

The partial derivatives:

$$\frac{\partial y}{\partial u} = v$$

$$\frac{\partial y}{\partial v} = u$$

The total variation of y depends on both the variations in u, v and thus,

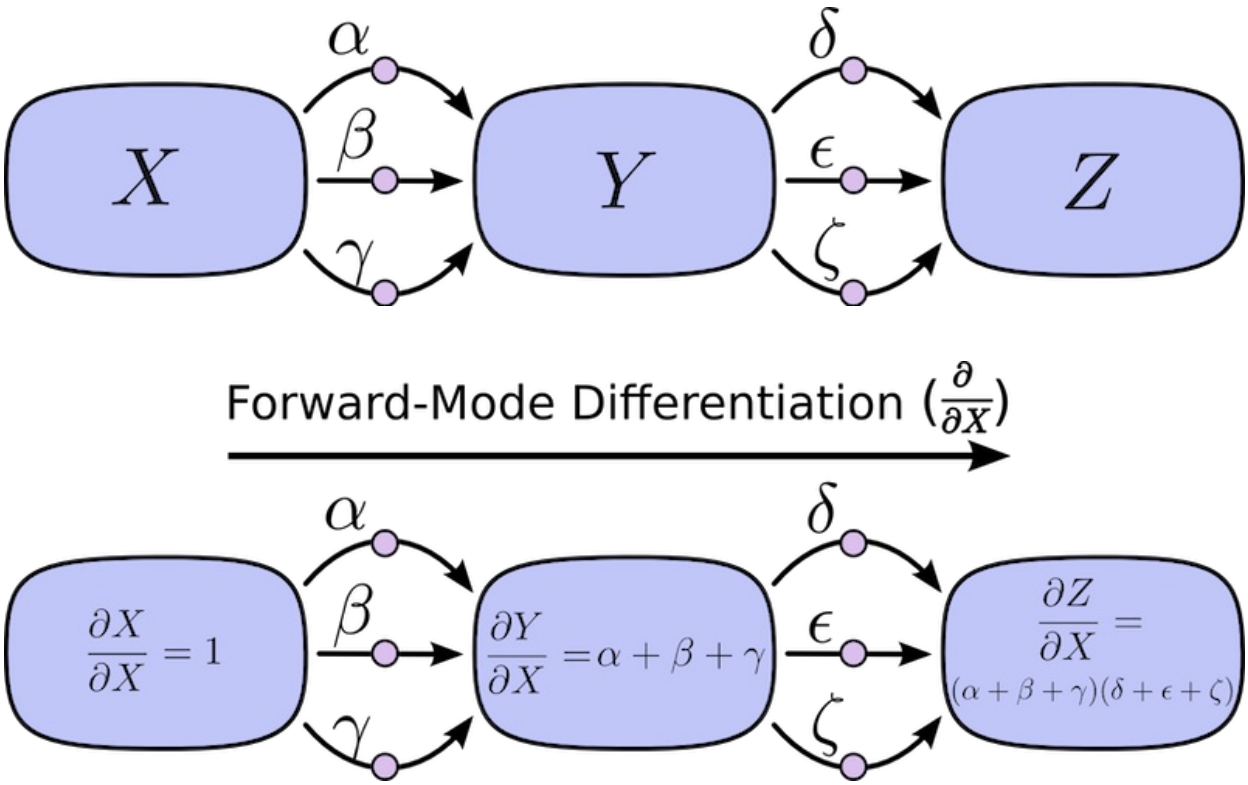
$$dy = \frac{\partial y}{\partial u} du + \frac{\partial y}{\partial v} dv$$

What this trivial example illustrates is that the derivative of a multivariate function is ultimately the addition of the partial derivatives and computations of its component variables. If a machine can compute any given sub-function as well as the partial derivative between any sub-functions, then the machine need only add-up the product of a function and its derivatives to calculate the total derivative.

An intuitive way of understanding automatic differentiation is to think of any complicated function as ultimately a a graph of composite functions. Each node is a primitive operation -- one in which the derivative is readily known -- and the edges on this graph -- the relationship of change between any two variables -- are partial derivatives. The sum of the paths between any two nodes is thus the partial derivative between those two functions (this a graph restatement of the total derivative via the chain rule).

Forward mode automatic differentiation thus begins at an input to a graph and sums the source paths. The below diagrams (from Christopher Olah's blog) provide an intuition for this process. The relationship between three variables (X, Y, Z) is defined by a number of paths (

$\alpha, \beta, \gamma, \delta, \epsilon, \zeta$). Forward mode begins with a seed of 1, and then in each node derivative is the product of the sum of the previous steps.



Consequently, provided that within each node there is an elementary function, the machine can track the derivative through the computational graph.

There is one last piece of the puzzle: dual numbers which extend the reals by restating each real as $x + \epsilon$, where $\epsilon^2 = 0$. Symbolic evaluation, within a machine, can quickly become computational untenable because the machine must hold in memory variables and their derivatives in the successive expansions of the rules of calculus. Dual numbers allow us to track the derivative of even a complicated function, as a kind of data structure that caries both the derivative and the primal of a number.

In our chain rule equation, there are two pieces to the computation: the derivative of the outer function applied to the inner function and that value multiplied by the derivative of the inner function. This means that the full symbolic representation of an incredibly complicated function can grow to exponentially many terms. However, dual numbers allow us to parse that symbolic representation in bitesized pieces that can be analytically computed. The reason for this is the Taylor series expansion of a function:

$$f(x+p) = f(x) + f'(x)p + \frac{f''(x)p^2}{2!} + \frac{f^{(3)}(x)p^3}{3!}$$

When one evaluates $f(x+\epsilon)$, given that $\epsilon^2 = 0$, then all the higher order terms drop out (they are 0) and one is left with $f(x) + f'(x)\epsilon$

To recap: automatic differentiation is an algorithmic means of computing complicated derivatives by parsing those functions as a graph structures to be traversed. Dual numbers are used as a sort of mathematical data structure which allows the machine to analytically compute the derivative at any given node. It is superior to analytic or symbolic differentiation because it is actually computationally feasible on modern machines! And it is superior to numerical methods because automatic differentiation is far more accurate (it achieves machine precision).

How to Use AutoDiff

How to Install

To install our package, clone the git repository using the following command line argument:

```
$ git clone https://github.com/VV-NS-CY-ZW-CS-207-Organization/cs207-FinalProject.g
```

Creating a Virtual Environment

After cloning the git repository, create a virtual environment to install all the necessary dependencies through the following command line arguments:

```
$ virtualenv env
$ source env/bin/activate
$ pip install -r requirements.txt
```

Import and Usage Example

In order to instantiate an auto-differentiation object from our package, the user shall first import the AutoDiff Driver from the AutoDiff library (see implementation section for more detail):

```
from autodiff.autodiff import AutoDiff as ad
```

The general workflow for the user is as follows:

- Instantiate all variables as AutoDiff objects.
- Input these variables into operators from the Operator class within the AutoDiff library to create more complex expressions that propagate the derivative.

The AutoDiff class is the core constructor for all variables in the function that are to be differentiated. There are two options for instantiating variables: Scalar and Vector, generated with create_scalar and create_vector respectively. Scalar variables have a single value per variable, while Vector variables can have multiple associated values. The general schematic for how the user shall instantiate AutoDiff objects is outlined below:

1. Create either a Scalar or Vector AutoDiff object to generate inputs to later pass into the function to be differentiated. The initialization works as follows:

```
x, y = ad.create_scalar(vals = [1,2])
z = ad.create_vector(vals = [1,2,3])
```

2. Next, the user shall import the Operator class and pass in these variables into elementary functions as follows:

```
from autodiff.operator import Operator as op
result = op.sin(x*y)
results = op.sin(z)
```

Simple operators, such as sums and products, can be used normally:

```
result = 6*x
results = z + 4
```

3. Finally, (as a continuation of the previous example), the user may access the value and derivative of a function using the .eval() method:

```
print(result.eval())
```

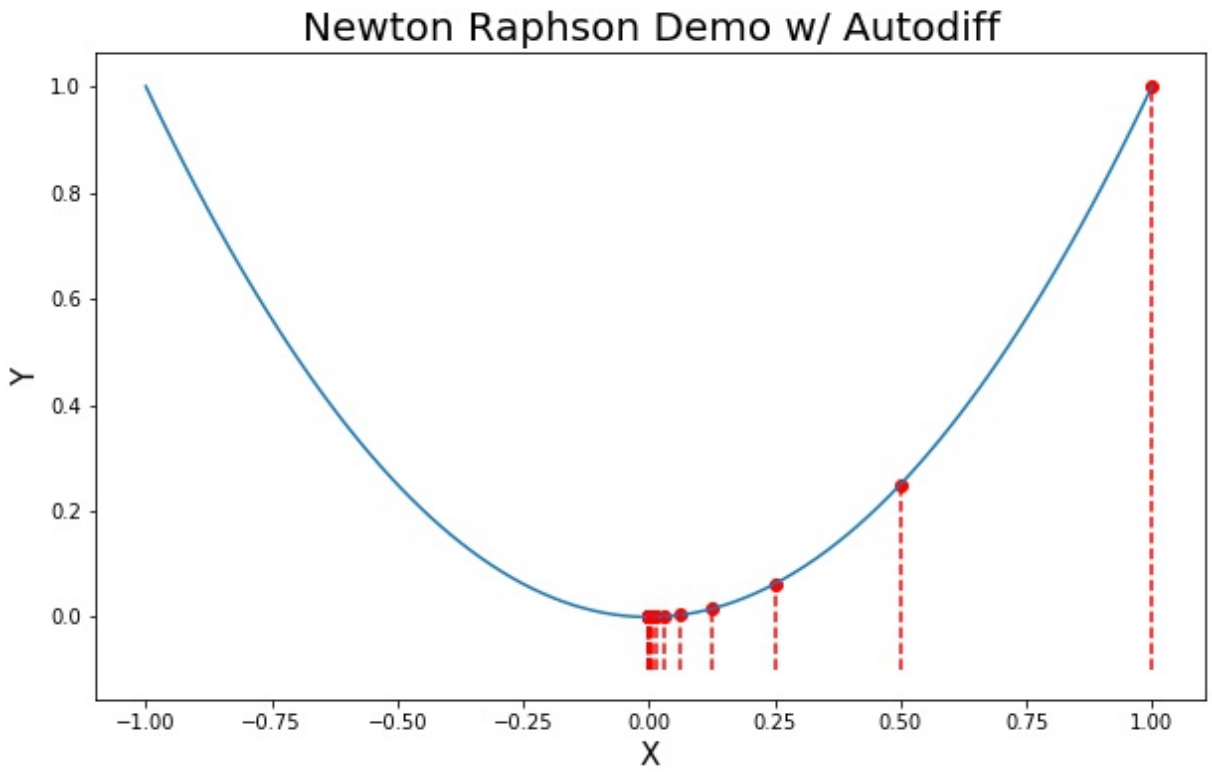
For scalars, result.eval() will return a tuple of (value, jacobian) where the jacobian is an array of partial derivatives of the function with respect to each scalar variable that was intialized at the

same time. For vectors, results.eval() returns a list of tuples (value, jacobian), with one tuple for each function in the vector.

Examples

There are two files in the top level directory of the autodiff package that demonstrate the usage of the package. The first is a file called "driver.py" which provides further examples on how the autodiff package can be used, and also serves as a comparison to numpy functions to prove its efficacy.

The second file is an interactive jupyter notebook which contains an example use case where the autodiff package performs well, namely, the Newton-Raphson method for approximating roots of functions. This notebook is titled "newton_demo.ipynb" and resides in "examples" folder in the top level directory of the package. The output of this demo is reproduced here for convenience:



Software Organization

Directory Structure

Our project will adhere to the directory structure outlined in the python-packaging documentation. At a high level (with exact names subject to change), the project will have the

following structure:

```
autodiff/
    autodiff/
        examples/
            __init__.py
            newton_demo.py
            ...
        nodes/
            __init__.py
            node.py
            scalar.py
            vector.py
        operators/
            __init__.py
            operator.py
        tests/
            __init__.py
            test_initializer.py
            test_newton.py
            test_opertor.py
            test_scalar.py
            test_vector.py
            ...
        autodiff.py  # driver for centralizing operator and node usage
    LICENSE
    MANIFEST.in
    README.md
    requirements.txt
    setup.py
    .gitignore
```

Modules

examples/

The Examples module will contain Python files with documented use cases of the library. Potential examples include an implementation of Newton's Method for approximating the roots of a non-linear function and a module which computes local extrema.

nodes/

The Nodes module will contain the *Node* superclass and associated subclasses (i.e., *Vector* and *Scalar*). Other packages and users will not use the *Node* class directly, instead they will work with its children through the *AutoDiff* driver.

operators/

The Operators module will contain the *Operator* class which will be imported by users directly.

tests/

The Tests module will contain the project's testing suite and will be formatted according to the pytest requirements for automatic test discovery.

Testing

Overview

The majority of the testing in this project's test suite will consist of unit testing. The aim is verifying the correctness of the application with thorough unit testing of all simple usages of the forward mode of automatic differentiation. Essentially, this will involve validating that our application produces correct calculations for all elementary functions. Additionally, a range of more complex unit testing will cover advanced scenarios such as functions with multidimensional domains and codomains as well as functions which created complexity via composition of elementary functions.

Additionally, the test suite will contain benchmarking and performance tests to demonstrate the scalability and limitations of the application.

Test Automation

In this project we will use continuous integration testing through Travis CI to perform automated, machine independent testing. Additionally, we will use Coveralls to validate the high code coverage of our testing suite (goal: 98-100%).
We will embed the Travis CI and Coveralls badges into the project README to provide transparency for users interacting with our project through GitHub.

Installation

To install our package, you can download the whole project from our github organization. You will get a folder named 'cs207-FinalProject-master' once you download the zip file. You can go into the folder to start your own script and use our library now. You can find 'driver.py' here, which is actually a use case and you may follow that to build up your own functions. It is worth noting that you have to put your script out of the 'autodiff' folder to import classes and functions from our package. When your script and the 'autodiff' folder is at the same level in the directory tree, you can use these sentences to import from ourt package:

```
from autodiff.autodiff import AutoDiff as ad
from autodiff.operator import Operator as op
```

User Verification

We plan on including all tests in the project distribution, thus allowing users to verify correctness for themselves using pytest and pytest-cov locally after installing the project package.

Distribution

Overview

Following the conventions defined in the python-packaging documentation, this project's packaging structure has been designed in such a way as to allow users to perform installation through pip. Specifically, our package will be uploaded to PyPi using Twine.

Licensing

This project will be distributed under the GNU GPLv3 license to allow free "as is" usage while requiring all extensions to remain open source.

Implementation

The purpose of this library (AutoDiff) is to perform automatic differentation on user defined functions, where the domain and codomain may be single- or multi-dimensional. At a high level, AutoDiff serves as a partial replacement for NumPy in the sense that rather than defining functions using NumPy methods such as *sin* and *cos*, the user can use the AutoDiff methods *sin* and *cos* that perform the same evaluation but also perform additional computation in the form of the derivative (i.e., thus implementing the forward mode of automatic differentiation).

To achieve this goal, the AutoDiff library implements the following abstract ideas:

1. Keeping track of the value and derivative of user defined expressions.
2. Allowing users to be as expressive as they would like by providing our own versions of binary and unary opertors.

With these goals in mind, the AutoDiff implementation relies on the

Nodes modules and the Operator class and allows user interface through the AutoDiff class which serves as a Node factory for initializing instances of *Scalar* and *Vector*.

Nodes

The Nodes module contains a *Node* superclass with the following basic design:

```
class Node():
    def eval(self):
        '''
        For the Scalar and Vector subclasses, this function returns the node's
        value as well as its derivative, both of which are guaranteed to be
        up to date by the class' operator overloads.

        Returns (self._val, self._jacobian)
        '''
        raise NotImplementedError

    def __add__(self, other):
        raise NotImplementedError

    def __radd__(self, other):
        raise NotImplementedError

    def __mul__(self, other):
        raise NotImplementedError

    def __rmul__(self, other):
        raise NotImplementedError

    ... # Additional operator overloads
```

Essentially, the role of the *Node* class (which in abstract terms is meant to represent a node in the computational graph underlying the automatic differentiation of the user defined expression) is to serve as an interface for the two other classes in the Nodes package: *Scalar* and *Vector*. Each of these subclasses implements the required operator overloading as necessary for scalar and vector functions respectively (i.e., addition, multiplication, subtraction, division, power, etc.).
This logic is separated into two separate classes to provide increased organization for higher dimensional functions and to allow class methods to use assumptions of specific properties of scalars and vectors to reduce implementation complexity.

Both the *Scalar* class and the *Vector* class have *_val* and *_jacobian* class attributes which allow for automatic differentiation by keeping tarck of each node's value and derivative.

Scalar

The *Scalar* class is used for user defined one-dimensional variables. Specifically, users can define functions of scalar variables (i.e., functions defined over multiple scalar variables with a one-dimensional codomain) using instances of *Scalar* in order to simultaneous calculate the function value and first derivative at a pre-chosen point of evaluation. Objects of the *Scalar* class are initialized with a value (i.e., the point of evaluation) which is stored in the class attribute self._val (n.b., as the single underscore suggests, this attribute should not be directly accessed or modifief by the user). Additionally, *Scalar* objects – which could be either individual scalar variables or expressions of scalar variables – keep track of their own derivatives in the class attribute self._jacobian. This derivative is implemented as a dictionary with *Scalar* objects serving as the keys and real numbers as values. Note that each *Scalar* object's _jacobian attribute has an entry for all scalar variables which the object might interact with (see AutoDiff Initializer section for more information).

Users interact with *Scalar* object's in two ways:

1. eval(self): This method allows users to obtain the value and derivative for a *Scalar* object at the point of evaluation defined when the user first initialized their *Scalar* objects. Specifically, this method returns a tuple of self._val and self._jacobian.
2. partial(self, var): This method allows users to obtain a partial derivative of the given *Scalar* object with respect to var. If self is one of the *Scalar* objects directly initialized by the user (see AutoDiff Initializer section), then partial() returns 1 if var == self and 0 otherwise. If self is a *Scalar* object formed by an expression of other *Sacalar* objects (e.g., self = Scalar(1) + Scalar(2)), then this method returns the correct partial derivative of self with respect to var.

Note that these are the only methods that users should be calling for *Scalar* objects and that users should not be directly accessing any of the object's class attributes.

Currently, *Scalar* objects support left- and right-sided addition, subtractionm multiplication, division, exponentiation, and negation.

Vector

Vector is a subclass of *Node*. Every vector variable consists of a 1-d numpy array to store the

values and a 2-d numpy array to store the jacobian matrix.
User can use index to acess specific element in a *Vector* instance. And operations between elements in the same vector instance and operations between vectors are implemented by overloading the operators of the class.
For *Vector* class, the elementary functions such as exponential functions and trig functions have not been implemented in the operator class yet. (But basic operations such as '+', '-', '', '/', '*' are supported for *Vector* class now.)

AutoDiff Initializer

The AutoDiff class functions as a Node factory, allowing the user to initialize variables for the sake of constructing arbitray functions. Because the Node class serves only as an interface for the *Scalar* and *Vector* classes, we do not want users to instantiate objects of the *Node* class directly. Thus, we define the *AutoDiff* class in the following way to allow users only to initialize *Scalar* and *Vector* variables:

```
from autodiff.nodes.scalar import Scalar
from autodiff.nodes.vector import Vector

class AutoDiff():
    def __init__(self):
        pass

    @staticmethod
    def create_scalar(vals):
        '''
        @vals denotes the evaluation points of variables for which the user
        would like to create Scalar variables.  If @vals is a list,
        the function returns a list of Scalar variables with @vals
        values.  If @vals is a single value, the user receives a single Scalar
        variable (not as a list).  This function also initializes the jacobians
        of all variables allocated.
        '''
        pass

    @staticmethod
    def create_vector(vals):
        '''
        The idea is similar to create_scalar.
        This will allow the user to create vectors and specify initial
        values for the elements of the vectors.
        '''
        pass
```

Using the *create_scalar* and *create_vector* methods, users are able to

initialize variables for use in constructing arbitrary functions. Additionally, users are able to specify initial values for these variables. Creating variables in this way will ensure that users are able to use the AutoDiff defined operators to both evaluate functions and compute their derivatives.

Variable Universes

The implementaiton of the AutoDiff library makes the following assumption: for each environment in which the user uses autodifferentiable variables (i.e., *Scalar* and *Vector* objects), the user initializes all such variables with a single call to create_scalar or create_vector. This assumption allows *Scalar* and *Vector* objects to fully initialize their jacobians before being used by the user. This greatly reduces implementation complexity.

This design choice should not restrict users in their construction of arbitrary functions for the reason that in order to define a function, the user must know how many primitive scalar variables they need to use in advance. Realize that this not mean that a user is prevented from defining new Python variables as functions of previously created *Scalar* objects, but only that a user, in defining a mathematical function f(x, y, z) must initialize x, y, and z with a single call to create_scalar. It is perfectly acceptable that in the definition of f(x, y, z) a Python variable such as a = x + y is created. The user is guaranteed that a.eval() and a.partial(x), a.partial(y), and a.partial(z) are all well defined and correct because a in this case is an instance of *Scalar*; however, it is not a "primitive" scalar variable and thus the user could not take a partial derivative with respect to a.

Operator

The *Operator* class defines static methods for elementary mathematical functions and operators (specifically those that cannot be overloaded in the *Scalar* and *Vector* classes) that can be called by users in constructing arbitrary functions. The *Operator* class will import the Nodes module in order to return new *Scalar* or *Vector* variables as appropriate. The design of the *Operator* class is as follows:

```
import numpy as np
from autodiff.nodes.scalar import Scalar

class Operator():
    @staticmethod
```

```
    def sin(x):
        pass

    @staticmethod
    def cos(x):
        pass

    ... # Other elementary functions
```

For each method defined in the *Operator* class, our implementation uses ducktyping to return the necessary object. If user passes a *Scalar* object to one of the methods then a new *Scalar* object is returned to the user with the correct value and jacobian. On the other hand, if the user passes a Python numeric type, then the method returns the evaluation of the corresponding NumPy method on the given argument (e.g., op.sin(1) = np.sin(1)).

Further Implementation

At this point, the following features are in progress but not yet completed:

1. Some operators still need to be overloaded for the *Vector* class (e.g., power)
2. Full support for create_vetor is not yet complete.
3. The *Operator* class currently only provides support for *Scalar* objects. We will provide support for *Vector* objects going forward.

External Depencies

This project aims to restrict dependencies on third-party libraries to the necessary minimum. Thus, the application will be restricted to using NumPy as necessary for mathematical computation (e.g., trigonometric functions). The test suite will use pytest and pytest-cov to perform unit testing and coverage analysis of such testing.

Future Plans

Our group will implement two future developments. The first we will signpost as dynamic variable universes. The current build requires the user to create a single universe of all variables within a given function and then creates a data structure to hold the associated Jacobian matrix of partial derivatives. In the future, users will be able to create additional variables that will communicate with previously defined functions. This is useful for optimization projects which involve the creation of additional features.

Speaking of optimization, we will also implement backwards auto-differentiation. Forward auto-differentiation computes the computational graph beginning with the input functions, while backpropagation (a special case of auto-differentiation) begins with the output and optimizes the function in reverse. In massively complex neural networks with many many inputs but only a few outputs, this greatly reduces computational time.

Automatic differentiation in machine learning: a survey. Baydin et al. 2018. Available at https://arxiv.org/abs/1502.05767