autograd Documentation

Release 1.0.0

Paxton Maeder-York, Adam Nitido, Dylan Randle and Simon Sebb

Jan 01, 2019

Contents

Author Paxton Maeder-York, Adam Nitido, Dylan Randle and Simon Sebbagh

Date Jan 01, 2019

Version 1.0

Autograd is a forward and reverse mode **Automatic Differentiation** (**AD**) software library. Autograd also supports optimization.

To install the latest release, type:

```
pip install dragongrad
```

See the *Installation notes* for details.

CHAPTER 1

Contents

1.1 Introduction

Autograd is an **Automatic Differentiation** (**AD**) software library.

Differentiation is a fundamental mathematical operation that underpins much of science and engineering. Differentiation is used to describe how a function changes with respect to a specific variable. Differential equations are common throughout science and engineering; from modeling the evolution of bacteria to calculating rocket thrust over time to predictive machine learning algorithms, the ability to rapidly compute accurate differential equations is of great interest.

The symbolic derivative of a function is precise; however as the function of interest become more complex, the symbolic derivative becomes increasingly difficult to determine. Numeric methods can be used to compute the derivative of such functions. The finite difference approach uses the definition of a derivative to estimate the derivative of a function; however, it suffers from low accuracy and instability.

AD is able to compute an approximation of the derivative of a function, **without computing a symbolic expression** of the derivative and with **machine precision** accuracy.

AD has many applications across Science and Engineering, the most popular one these days being Deep Neural Networks. These models try to fit a function with >*10M* parameters to a dataset. To do so, they use Gradient Descent algorithms using gradient approximations provided by AD. Famous applications include **Alpha Go**, **Self-Driving Cars** and **Image Recognition**.

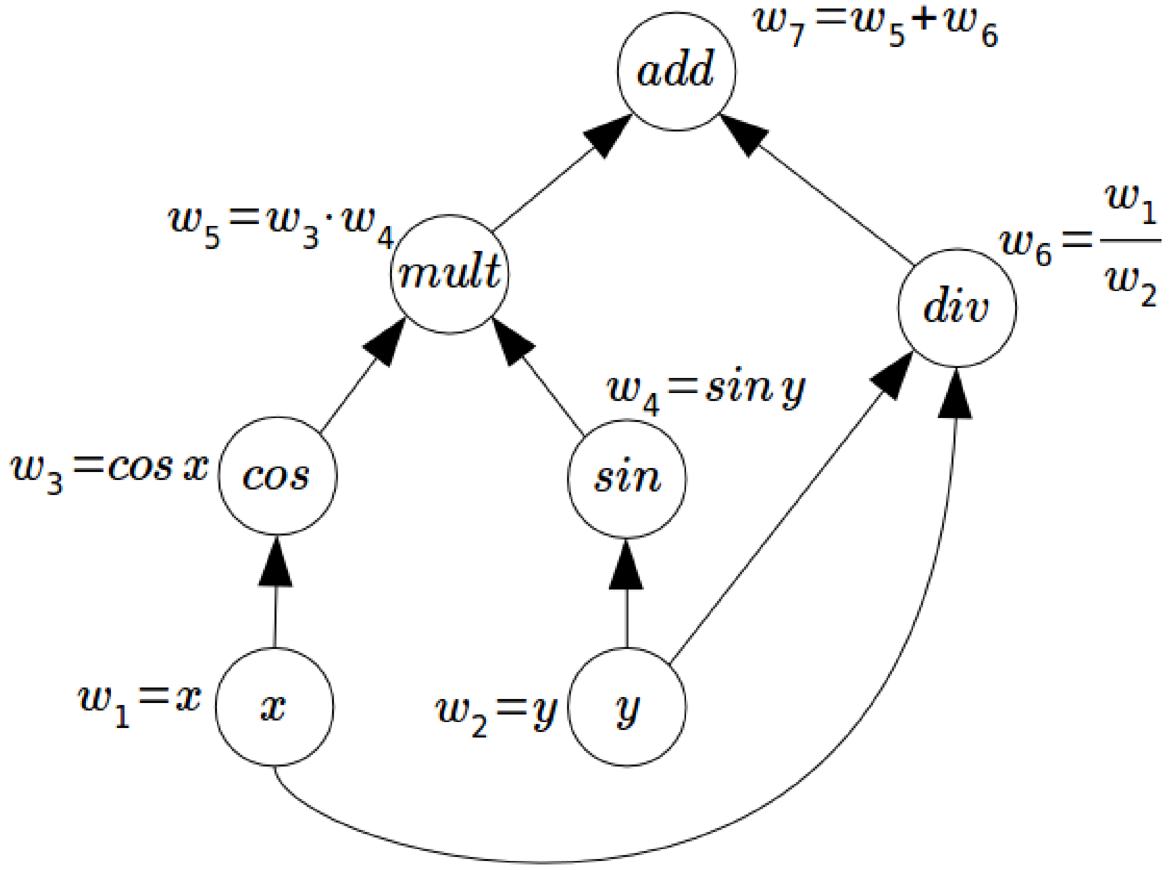
1.2 Background

The basic idea that underpins the AD algorithm is the chain rule:

Essentially what the algorithm does is take a complex function and rewrite it as a composition of elementary functions. Then, using stored symbolic derivatives for these elementary functions, the algorithm "reverse expands" the chain rule by starting with the innermost function and building on it. This means that the gradient of the innermost function will be computed and passed through and computed in each other function until reaching the original function.

In other words, we will represent a function whose derivative we wish to compute by a "computational graph" which builds up some set of operations sequentially. In the computational graph, each note is a basic operation and the edges pass information through the nodes. In the computational graph, the data passed through the nodes contains the value of the original function and the gradient evaluated at some value.

An example of a computational graph is:



In the example above, the *w1* node contains the gradient for input *x* at some value, the data from *w1* is then passed through the *cos()* operation to create *w3*. *w3* is later multiplied with *w4* to create *w5*, and so on. We will pass our input value along the "trace", and by judicious application of the chain rule, we will compute the derivative of the overall function. The traces can be through of as the ordered set of operations that the data undergoes.

1.3 Installation

Autograd can be install though pypi and from our GitHub repository. The recommended way to install autograd is through pypi.

1.3.1 Pypi installation

Pypi installation:

```
pip install dragongrad
```

1.3.2 GitHub Installation

1. Create a virtual environment:

```
cd my_directory
virtualenv my_env
```

2. Activate the virtual environment:

```
source my_env/bin/activate
```

3. Download Package from GutHub (or clone) and Unzip:

```
unzip cs207-FinalProject-master.zip
```

4. Install Dependencies using Pip:

```
pip install -r cs207-FinalProject-master/requirements.txt
```

5. Install autograd – this step is **Very Important**:

```
cd cs207-FinalProject-master
python3 setup.py install
```

1.3.3 Requirements

autograd works with Python3.

Both installation methods will install the correct version of Numpy, It is recommended install this software in a virtual environment.

Dependencies - Numpy.

1.4 Autograd Usage

Autograd comes with an user-friendly API, for both forward and reverse mode.

1.4.1 General rules

Autograd will nearly always give you a result. However, in order to ensure that you compute what you exactly think you are computing, please make sure to read carefully these points :

1. When you define a Variable, it is automatically set as the input node of the computational graph
2. Thus, if you define two variables like `x=Variable(3)` and then `y=Variable(4)`, the input node of the graph will be y only, and you will not compute gradients with respect to x. Never.
3. If you want to work on function of several inputs, please refer to the section *Multiple Inputs*
4. Before you try to access the `variable.gradient` attribute, you should run `variable.compute_gradients()`.
5. When you are in reverse mode, don't forget to reset the computational graph when you are running a new function call. You can do it with `ad.reset_graph()`
6. Enjoy! :)

Additional resources are available in the Demo_Notebook - make sure to have matplotlib installed if you want to run the Demo_Notebook

1.4.2 Simple Differentiation Case

Example: How to differentiate `f(x) = sin(x) + cos(x)` at x = pi:

```
>>> import numpy as np
>>> import autograd as ad
>>> from autograd.variable import Variable
>>> ad.set_mode('forward')
>>> x = Variable(np.pi)
>>> b1 = ad.sin(x)
>>> b2 = ad.cos(x)
>>> b3 = b1 + b2
>>> print(b3.gradient)
-1
```

b3 will contain the gradient of `y = sin(x) + cos(x)` at x = pi

Example: How to differentiate `f(x)=sin(cos(x+3)) + e^(sin(x)^2)` at x = 1:

```
>>> import numpy as np
>>> import autograd as ad
>>> from autograd.variable import Variable
>>> ad.set_mode('forward')
>>> x = Variable(1)
>>> b1 = x + 3
>>> b2 = ad.sin(x)
>>> b3 = ad.cos(b1)
>>> b4 = ad.sin(b3)
>>> b5 = b2*b2
>>> b6 = ad.exp(b5)
>>> b7 = b6 + b4
>>> print(b7.gradient)
2.44674864
```

b7 will contain the gradient of `f(x)=sin(cos(x+3)) + e^(sin(x)^2)` at x = 1

1.4.3 Differentiation of Functions

If a user wants to differentiate multiple values is recommended that users create functions that wrap around autograd:

```
def function(x):
    x1 = av.Variable(x)
    b1 = ad.sin(x1)
    b2 = ad.cos(x1)
    b3 = b1 + b2
    b3.compute_gradients()
    return(b3.data,b3.gradient)
```

This function can be used to loop and differentiate values:

```
value = list()
data = list()
```

(continues on next page)

(continued from previous page)

```
gradient = list()
for i in np.linspace(-2 * np.pi, 2 * np.pi):
    value.append(i)
    output = function(i)
    data.append(output[0])
    gradient.append(output[1][0])
```

1.4.4 Multiple Inputs

As this package handles vector to vector mapping, we can theoretically consider every function of several variables as a function of vector input. For exemple, we can see the function f(x,y,z) as a function of 3 variables which are scalar, but also as a function of one variable, which is a vector of R3. We refer to these methods to the Multiple Variables Mode and Vector Mode, respectively.

Vector Mode

Before performing any operations, you should embbed the inputs of your function in one big variable

```
def vector_function(x,y):
    big_variable = Variable([x,y])
    x,y=big_variable[0], big_variable[1]

    b1 = ad.exp(-0.1*((x**2)+(y**2)))
    b2 = ad.cos(0.5*(x+y))
    b3 = b1*b2+0.1*(x+y)+ad.exp(0.1*(3-(x+y)))


    b3.compute_gradients()
    return(b3.data,b3.gradient)
```

In that case, you will have *b3.gradient* as a matrix of shape 1*3, because you considered the function as a vector function mapping from R3 to R.

Multiple Variables

Pass multiple variables:

```
def vector_function(x,y):
     x,y=av.Variable.multi_variables(x,y)

     b1 = ad.exp(-0.1*((x**2)+(y**2)))
     b2 = ad.cos(0.5*(x+y))
     b3 = b1*b2+0.1*(x+y)+ad.exp(0.1*(3-(x+y)))


     b3.compute_gradients()
     return(b3.data,b3.gradient)
```

In that case, we have `b3.gradient = [grad(b3, x), grad(b3, y)]` with `grad(b3,x)` refers to the gradient of the function `x-->b3` evaluated at x.

The choice of which mode is up to you, the multi_variables is useful when you deal with several inputs with different shapes:

```
def vector_function(x,L,N):
     x, L, N = av.Variable.multi_variables(x,L, N)

     b1 = ad.sum_elts(L)
     b2=x*L
     b3=x+b2
     b4=N*L
     b5=b3+b4[0]

     b5.compute_gradients()
     return(b5.data,b5.gradient)
```

We will then have `b5.gradient = [grad(b5,x), grad(b5,L), grad(b5,L)]` with `grad(b5, L)` a matrix of shape 1*dim(L), etc. . .

This method is quite straightforward and intuitive, not as what we would have had to do in the vector mode to get the gradients of x and L separately

```
gradient_b5_x = b5.gradient[:,0:1]
gradient_b5_L = b5.gradient[:,1:dim(L)+1]
gradient_b5_N = b5.gradient[:,dim(L)+1:]
```

with even more complicated gradient extractions when you have more input vectors of different sizes.

The performance of these two methods is identical.

1.4.5 Forward or Reverse Mode

Forward mode is set by default, but to explicitly set forward mode:

```
>>> import autograd as ad
>>> ad.set_mode('forward')
```

Reverse mode can be easily set:

```
>>> import autograd as ad
>>> ad.set_mode('reverse')
```

Once reverse mode is set, all differentiation in the session will be calculated in reverse mode, unless forward mode is explicitly set.:

```
>>> import autograd as ad
>>> ad.set_mode('reverse')
>>> ad.set_mode('forward')
```

The resulting setting is forward mode

1.4.6 Optimization

Currently, autograd supports gradient descent and Adam optimization, in both forward and reverse mode.

Optimization Setup:

```
import numpy as np
import autograd as ad
from autograd.variable import Variable

#set to forward mode
ad.set_mode('forward')

#define function
def loss(params):
    var = Variable(params)
    x,y = var[0], var[1]
    l = (x+5)**2 + (y+3)**2

    l.compute_gradients()

    return (l.data, l.gradient)
```

1.4.7 Gradient Descent

Autograd has implemented Gradient Descent.

Gradient Descent Optimization:

```
#import gradient descent
from autograd.optimize import GD

#initialize values
x_init = [10, 4]

#create optimization object and set parameters
optimize_GD = GD(loss, x_init, lr=0.1, max_iter=1000, tol=1e-13)

#solve
sol = optimize_GD.solve()
```

1.4.8 Adam

Autograd has implemented the Adam Optimizer: Adam: A Method for Stochastic Optimization.

Adam Optimization:

```
#import Adam Optimizer
from autograd.optimize import Adam

#initialize values
x_init = [10, 4]

#create optimization object and set parameters
adam = Adam(loss, x_init, lr=0.1, max_iter=1000, tol=1e-13)

#solve
sol = adam.solve()
```

1.5 Software Organization

The *autograd* package organized into various modules. Our basic directory structure will look as follows:

```
cs207-FinalProject/
    autograd/
        __init__.py
        blocks/
            __init__.py
            block.py
            expo.py
            hyperbolic.py
            operations.py
            trigo.py
        tests/
            __init__.py
            test_basic.py
            test_autograd.py
            ...
        config.py
        node.py
        utils.py
        variable.py
        optimize.py
    docs/
        dev_milestones/
            milestone1.md
            milestone2.md
        ...
    README.md
    requirements.txt
    setup.py
    Demo_Notebook.ipynb
```

The autograd package is organized into a few key modules:

- `block.py`: objects implementing the core computational units of the graph, namely `data_fn` (*f(x)*) and `gradient_fn` (*f'(x)*).
- Within the blocks submodule, there additional block operations - categorized by operation type.
- `variable.py`: data structure containing the function value and gradient value
- `utils.py`: general utility functions that are reused throughout the project
- `optimize.py`: contains the optimizer classes and functions
- `node.py`: contains the node class and computational graph class for reverse mode
- `config.py` : Stores all the nodes for reverse mode
- `tests`: contain all the tests, divided by which module is being tested
- `docs`: contains development milestones in a sub directory, also contains useful information about the project, hosted on read the docs.

1.6 Implementation

Recalling the background section, we saw that the automatic differentiation framework splits a complex function into several atomic functions which derivative is easy to compute. Then, the results are aggregated using the chaing rule.

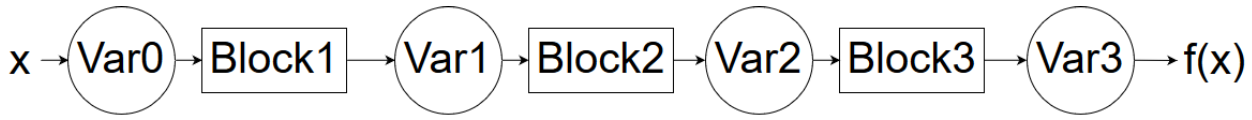
This package has been designed so that it is easy for a new user to define his own new atomic function. For instance, we did not implement convolution operations over vectors, but a new user could easily define it, following the API we will describe.

Important : The functionment of the package is slightly different depending on wether you use the forward or the reverse mode. In this optic, we will first present the forward mode and then highlight the differences happening in the reverse mode.

1.6.1 Forward Mode

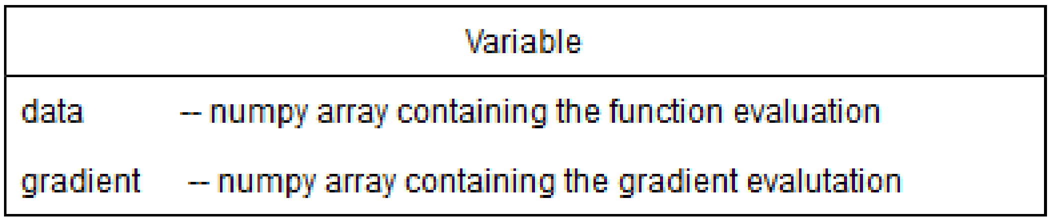
The core data structures in this package are `Variables` and `Blocks`.

We are going to consider that every function can be split into core atomic functions, each of which we will call a *Block*. Thus, the application of a function is a mere composition of *Block* operations to *Variables*.



Variable

The first core data structure is *Variable*. This object will flow through several *Blocks*, storing the new values of the functions computed, as well as the gradient computed so far.



Variable	
data	-- numpy array containing the function evaluation
gradient	-- numpy array containing the gradient evalutation

It contains two main attributes : `data` and `gradient`. The *data* attribute stores the value of the function computed so far. The *gradient* attribute contains **the value of the derivative of this node with respect to the input node**. It is defined for every variable but note that in the reverse mode, the *gradient* attribute is only accessible on the output node, we'll develop this further.

In each block, the input `Variable` brings the information from the previous data and gradients computed and propagates the data and gradient flow forward.

For exemple, taking the previous example : Var2.data will be the numpy array resulting from the sequence of operations `Block2(Block1(Var.data))`

Samely, Var2.gradient will contain the gradient of the function `x-->Block2(Block1(x))` evaluated at the point x=Var0.data

The main method of a variable is `.compute_gradients()` : it allows the *gradient* attribute of a variable to be propperly defined. In reverse mode, it triggers the reverse gradient flow, as we will see.

In forward mode, even if the gradients are computed on the fly, we need to call the compute_gradients() method, as it allows to handle the case where you have several input nodes. See below.

Initialization

This package handles vector functions, meaning that it can compute gradients of function from Rn to Rp. Hence, the .gradient attribute is not a gradient, but rather a Jacobian matrix.

Meaning, if we assume that Var0.data is an array of shape n and Var3.data is an array of shape p, then Var3.gradient will be a matrix of shape p*n

The basic initializer for that class is :

`def __init__(self,data, gradient=None, constant=False, input_node=True):`

The *data* argument is either a scalar or a list/np.array that refers to the point we wish to evaluate the function.

The *gradient* argument is used to set the gradient of this variable when we initialize it, it is used later with the *Blocks*.

Attention : before tring to access the *.gradient* of a Variable, you should always call `my_var.compute_gradients()`

The *constant* argument allows to indicate if we are dealing with an actual *Variable* or if this is just a *Constant*. See the Constant section for more explanation

The *input_node* argument is used to specify if the Variable created is the input of a complex function. Meaning, when the user want to define a new function, he will define it as

```
def function(x):
 y=do_stuff(X)
 return(y)
```

thus, the *input_node* for this function is the input variable x. Note that when a user creates a new input node, it overwrites the older : you cannot have several input nodes defined with several *Variable(*args)* calls. To manage several inputs, check the following sections.

If nothing is indicated by the user, the default value of `Variable.gradient` is an Identity matrix, meaning we are at the beginning of the computational graph : the jacobian of a variable with respect to itself is the Identity matrix, with corresponding dimensions.

The constants are managed as Variables with a initial `gradient` as a matrix of 0's. See below.

Constant

A *Constant* object is just meant to embed the notion of constants in the operations we encounter. For instance, if you want to compute the gradient of `f(x)=7*x+3`. We will not compute derivatives with respect to *7* or *3* which would not make sense. Rather, we embed the constants in the function within this class.

A *Constant* is a subclass of *Variable* but it is always initialized with a *gradient* attribute as a Jacobian of 0's. This way, we ensure that this constant does not participate in the gradient computation.

The reason why we decided to embed these constants as variables, is because it allows to have a unified API for these two objects. The difference is that constants are used in the data flow but not in the gradient flow. Also, a *Constant* cannot be the input node of the computational graph, obviously.

Multiple Variables

As this package handles vector to vector mapping, we can theoretically consider every function of several variables as a function of vector input. For exemple, we can see the function `f(x,y,z)` as a function of 3 variables which are scalar, but also as a function of one variable, which is a vector of R3.

Thus, if you want to opt for the vector approach, you will have to process as follows :

- vector approach

In this approach, you define one big input node that embeds all your input variables

```
def f(x,y,z):
 vector_variable=Variable([x,y,z])  #create the vector variable with the data of x,y␣
↪and z

 #extract the relevant variables
 #the [] operator extracts both data and gradient and create a new corresponding␣
↪variable
 x_var, y_var, z_var = vector_variable[0], vector_variable[1], vector_variable[1]

 output=do_stuff(x_var, y_var, z_var)
 return(outpput)
```

Let's assume that the output of this function is a scalal, this way you will compute the gradient of f as a function from R3 in R and the gradient of *output* will be a Jacobian matrix of shape 1*3.

Then, if you are in an optimization framework, you will have to extract the gradients of *output* with respect to each input respectively. Namely, you will want to perform the update

```
x <--x + lr* grad(output, x)
y <--y + lr* grad(output, y)
z <--z + lr* grad(output, z)
```

but you have to extract the gradients from the jacobian matrix :: #never forget to compute_gradients() before trying to access to the gradient of a variable output.compute_gradients() grad(output, x) = output.gradient[0,0] grad(output, x) = output.gradient[0,1] grad(output, x) = output.gradient[0,2]

or perform that update in a vectorized fashion : `vector_of_inputs += lr * output.gradient[0]`

- distinct inputs approach

The other way to look at it is to say, that f has 3 input variables, so in our framework, the computational graph will have 3 input nodes.

Disclaimer : when you define a new *Variable* it overwrites the current input node of the graph, so you should **not** process like

```
x_var = Variable(x)
y_var = Variable(y)
z_var = Variable(z)
```

If you do this, the input node of the graph will be z_var. . .

To tackle this, you will use the **classmethod** of Variable :

`x_var, y_var, z_var = Variable.multi_variables(x,y,z)`

This function defines several input variables, and set them as input nodes of the graph. Then the program runs as usual, with one difference : still with the previous example, the function f will have 3 inputs and not one big vector input

Hence, *output.gradient* will be equal to the **list** of the gradients of f with respect to all the variable **in the same order they have been defined**. Namely

```
output.compute_gradients()
# we have : output.gradient = [grad(output, x), grad(output, y), grad(output, z)]
```

with `grad(output, x)` an array of shape 1*1. If f had an output dimension of p, we would have `grad(output, x)` as a matrix of shape p*1.

In this exemple, I took x, y and z as scalars, but you could totally define a function like

```
def f(x, L):
 x_var, L_var = Variable.multi_variables(x,L)
 ...
```

With x a scalar and L a list of size n.

In this context of multi_variables, we basically create one big variable that aggregates all the individual inputs and then extract them as variables, it also sets these variabales as the input nodes of the computational graph . This process allows to define one single input variable while defininig several input nodes.

In forward mode, it is useful as when we call *compute_gradients*, we will return the list of the gradients of the output node w.r. all the single input variables. We thus need to know which are the input nodes and in which order they have been defined. This *multi_variables* function allows to do this.

In reverse mode, it is also useful to define the input nodes of the computational graph.

Block

The second core data structure is the `Block`. It is an atomic operation performed on `Variable`. For instance, sin, exp, addition or multiplication. for flexibility of the code, we implemented a generic *Block* type as well as a more specific one : the *SimpleBlock*.

In *Autograd*, all the blocks stand for functions : we have the sinBlock, the cosBlock, . . . , and also the extractBlock that overrides the [] method. . .

Thus, before calling a function on a variable, we need to instantiate the corresponding block and then call it

```
from autograd.blocks.trigo import sin
from autograd.variable import Variable

x= Variable(3)
sinBlock=sin()
y=sinBlock(x)
```

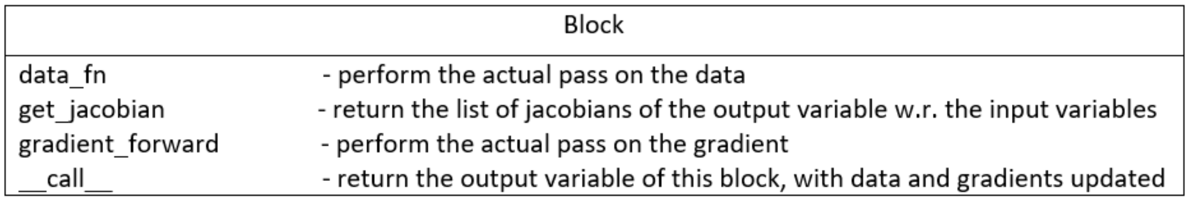
However, in order to have a better user experience, we instantiate all the blocks in the *__init__.py* of *Autograd* so that the user can directly have access to these blocks

```
from autograd.variable import Variable

 x= Variable(3)
 y=ad.sin(x)
```

We will describe the different blocks we have but all of them work as follows : It takes one or several input variables and then tt outputs a new Variable with updated data and gradient.

Main Block



Block	
data_fn	- perform the actual pass on the data
get_jacobian	- return the list of jacobians of the output variable w.r. the input variables
gradient_forward	- perform the actual pass on the gradient
__call__	- return the output variable of this block, with data and gradients updated

In forward mode, the `Block` contains four major methods that we will describe :

- data_fn

It is used to define the function evaluation for that block. For example in the *additionBlock*, we coded

```
class add(Block):
  """
  addition of two vector inputs
  """
  def data_fn(self, *args):
    #returns the data of the output variable of this block
    new_data = np.add(args[0].data, args[1].data)
    return(new_data)
```

This method is specific to each block

- get_jacobians

Every block defines an atomic function. The *get_jacobian* method returns the jacobian of this atomic function w.r to all its inputs separately. For example, still in the *additionBlock*

```
class add(Block):
    """
    addition of two vector inputs
    """
    def data_fn(self, *args):
        new_data = np.add(args[0].data, args[1].data)
        return(new_data)

    def get_jacobians(self, *args):
        shape=args[0].data.shape[0]
        first_term = np.eye(shape)
        second_term = np.eye(shape)

        return([first_term, second_term])
```

In fact, when we have `z=x+y` we have grad(z, x) as the Identity matrix with corresponding shape. Samely for grad(z, y)

This method is specific to each block

- gradient_forward

Is used to propagate the gradient flow forward : it takes the gradients of the input variables of the block, multiply them with the jacobians of this bloc, thanks to the *.get_jacobians()* method. And then it outputs the gradient of the output variable

```
class Block():
  def gradient_forward(self, *args, **kwargs):
    #concatenate the input gradients
    input_grad = np.concatenate([var.gradient for var in args], axis=0)

    #concatenate the jacobians of the block
    jacobians = self.get_jacobians(*args, **kwargs)
    jacobian = np.concatenate([jacob for jacob in jacobians], axis=1)

    #combine the gradients of the input variables with the jacobians of the block
    new_grad = np.matmul(jacobian, input_grad)

    return(new_grad)
```

This method is common to all the blocks

Explanation :

Let's consider a computational graph which transforms : `x = x_0 --SINBLOCK--> x_1 --COSBLOCK--> x_2 --EXPBLOCK--> x_3 = f(x)`

As previously stated, the variable x_0 has the default value for `gradient`, which is the identity matrix. with gradient_forward, the SINBOCK will output a variable which has a data of `sin(x_0.data)` and a gradient of `cos(x_0.data) * x_0.gradient`.

Then, COSBLOCK will output a variable with data = `cos(x_1.data) = cos(sin(x_0.data))` and gradient = `-sin(x_1.data) * x_1.gradient`, and we will have

`x_2.gradient = jac_COSBLOCK * jac_SINBLOCK * x_0.gradient`

This is how the gradients flow in the forward mode.

- __call__

take as input one or several variables, perform a forward pass on data and gradient and return a new output variable.

`new_var = block(input_var_1, input_var_2)`

No storing of the computational graph

The solution we provided is efficient in that we don't store the computation graph in the forward mode. The values of the variables are computed on the fly, both data and gradient.

Usually, the user overwrite its variable so we have a minimal memory usage

```
import autograd as ad
from autograd.variable import Variable

x=Variable([34,54,65])
y=ad.sin(x)
y=ad.cos(y)
y=ad.exp(y)
for _ in range(12345):
  y *= 3

output = y+x
```

the variable y has been overwriten : in this sequence of operations, we have stored only 3 variables : x, y, and output.

If we were to store naively all the computational graph, we would have stored way more variables. . . .

Of course, the `autograd` package is being built respecting the design patterns for good development, the user will have the possibility to build his own *Block* if he would not find a specific function among the ones we provide. The user would have to follow the *Block* interface and provide a `data_fn` as well as a `get_jacobians`.

However sometimes, the block we want to implement is just a vectorized simple function. For instance, sin(x) applies sin(.) to all the elements of x.data. This leads to the useful subclass to handle vectorized functioons, the *SimpleBlock*

Simple Block

The simple block allows to represent simple functions : in the context of vector mapping, we usually have some functions that apply the same operations to all the elements. They are called vectorized functions.

For example, `sin(x) = [sin(elt) for elt in x.data]`

For these functions, which have only one input, the jacobian is easy to compute, it is equal to the diagonal matrix with the derivative of the block evaluated at the input points. In other words

```
``jacobian = np.diag(block.gradient_fn(input_variable))``
```

Thus, for this class we overwrite the *.get_jacobians()* as follows

```
def get_jacobians(self, *args, **kwargs):
        """
        get the Jacobian matrix of the simple block. It is a diagonal matrix easy to
↪build from the
        derivative function of the simpleBlock
        """
        #get the elements of the diagonal
        elts = self.gradient_fn(*args, **kwargs)
        jacobian = np.diag(elts)
        return([jacobian])
```

This is a method generic for all the simple blocks

We thus implement a *data_fn* as previously, but now, instead of defining a *get_jacobians()* method, we only need to define the derivative of the simple function, in a new method *gradient_fn()*. For example for the *SinBlock*

```
class sin(SimpleBlock):
    """
    vectorized sinus function on vectors
    """
    def data_fn(self, args):
        new_data = np.sin(args.data)
        return(new_data)

    def gradient_fn(self, args):
        grad = np.cos(args.data)
        return(grad)
```

The *gradient_fn()* method is specific to each block.

This elegant way to represent functions allows an easy definition of new blocks, but more : it allows the implementation of the reverse mode in an elegant fasion.

1.6.2 Reverse Mode

In the reverse mode, the gradients are not computed from the input nodes to the output nodes in the computational graph. Instead, they are computed from the output node to the input nodes.

The reverse mode applies a forward mode on the data, stores relevant information, and applies a reverse pass on the gradients.

To do this, we need to store all the intermediate values that have been used to compute the output variable.

We achieve this by doing the following modifications on the classes :

Variable

- gradient

This attribute is no more accessible to all the variables. The only variable that as a non‘None‘ gradient attribute is the output variable **after** having called `output_variable.compute_gradients()`

- .compute_gradients()

This method now applies the reverse pass to compute the gradients, it also allows to have access to the output_variable.gradient attribute

- node

We also introduce a new class for the reverse mode, the *Node*. We will describe it in the next section

Node

Previously, we were talking without distinction of nodes and variables. Now however, we will be very careful not to mix these two concepts.

A *Node* is a new separate class used in the reverse mode, that allows to store relevant information from the forward pass. Everytime a new Variable is created, a node is created, stored in a global buffer (*config* file), and is associated to the variable. A node has two main attributes : *gradient‘and ‘childrens* :

- gradient

It is used to store the gradient of the output variable w.r. this node’s variable. Meaning that `output_variable.node.gradient = Identity` and `input_variable.node.gradient` is actually the gradients we are looking to compute : it is the gradient of the function w.r. the input variables.

- childrens

list that store the nodes of the variables that have been used to compute this new node’s variable, and their respective gradient. Namely

x=Variable(2) y=sin(x) z=x+y

x is the input_node, his node’s children dictionnary is empty.

y’s node has one children : *x*’s node. Moreover, the transformation x–>y is associated with a `jacobian = cos(x.data)`. Thus, we will have `y.node.childrens=[{'node':x.node, 'jacobian':cos(x.data)}]`

z’s node has two childrens : *x*’s node and *y*’s node. Moreover, the transformation x,y–>z is associated with two jacobians

`jacobian_x = identity`

`jacobian_y = identity`

Thus, we will have `z.node.childrens=[{'node':x.node, 'jacobian':identity}, {'node':y.node, 'jacobian':identity}]`

The main method of *Node* is the *backward()* method :

It is used to compute **recursively** the gradients of the ouput variable w.r. to the input node.

To do this, it sets the gradient of the output node to the identity, and propagate backwards the gradients using the children's jacobians :

For each children node, it computes the contribution of this node to the output gradient, and updates the *gradient* of the children node

```
for child in self.childrens:
  node,jacobian=child['node'], child['jacobian']
  new_grad = np.dot(self.gradient, jacobian)
  node.update_gradient(new_grad)
```

This process is repeated until we computed the gradients of all the input nodes, they are the nodes for which `childrens=[]`.

At the end of this function call, all the nodes involved in the computational graph have a *gradient* attribute set.

Computational Graph

Main class that stores the information of the computational graph. It is defined in the `__init__.py` of *Autograd* so that we can access it anytime with `ad.c_graph`

Should be noted that as we store the dependencies among the nodes in the nodes themselves, we don't need to store them again in the computational graph. Meaning : every node define a tree with the *childrens* attribute, we only need to store the global informations about the computational graph :

- input_nodes

List that store the input nodes of the computational graph

- output_node

Store the output node of the computational graph

- input_shapes

List that store the shapes of the input variables. For example with `x, L, y, Z = Variable.multi_variables(x, L, y, z)`

we will have `ad.c_graph.input_shapes = [dim(x), dim(L), dim(y), dim(Z)]` (with dim(x) the dimension of the scalar/vector of x). This attribute is important only when dealing with several distinct inputs, when we need to reconstruct the several distinct gradients in the *compute_gradients()* call

Given these informations, we can compute the reverse pass on the gradients. Here is the event flow :

1. User calls `output_variable.compute_gradients()`
2. This function will first define the output node of the computational graph as the output_variable.node
3. given this output node, we make a first reverse pass to see which nodes have been used to compute this output_variable, and how many times.

For example, in the case

```
x=Variable(3)
y=sin(x)
output_variable=x+x
```

The define path will assess the several numbers

```
x.node.times_used = 2
y.node.times_used = 0
output_variable.node.times_used = 0
```

As the variable *y* did not contribute to the computation of the output node, and the output node has not been used to compute anyting.

4. We call *backward()* on the output variable node. This function will set the node's gradient of all the nodes selected in the *define_path()* call
5. If the computational graph has one input node, we return the gradient of this vector mapping. Is is a jacobian matrix. The *output_variable.gradient* attribute is set to this matrix, as in the forward mode.
6. If we have several input nodes (defined with multi_variables), we return the list of the jacobians defining the contribution of each of the input nodes. The *output_variable.gradient* attribute is set to this list of matrices, as in the forward mode.

- reset_graph

Eventually, when we want to re-run the function, we need to reset the graph : we zero the gradients, as well as the number of times the nodes have been used.

Note that when the user define a new Variable, it automatically sets this variable as input node of the graph. Thus, we can remove all the previously created nodes and restart from scratch : the buffer that store the nodes created is flushed. Thus, we cannot use the previously created variables, we need to recompute them.

Block

In the reverse mode, the only method modified is the *__call__* :

- The forward data pass is not modified, we create a new variable with corresponding updated data attribute
- The ouput variable is created with a node. We set this node's *childrens* using the jacobians of the block and the input variables' nodes

1.7 Notebooks

1.7.1 Demo Notebook

The demo notebook is hosted on our GitHub repository.

1.7.2 Converge Notebook

The convergence test notebook is hosted on our GitHub repository.

1.8 License

1.9 Future Work

The next step in this project is feed forward neural networks implementation.

In fact, we have developped all the tools required to train a neural net, we only need to specify the weights of the model as input nodes and implement new blocks : convolution and several other layers.

In terms of applications, we think about medical image processing : using convolutional neural networks, we can detect and segment some regions on a medical X-ray. This tool can help the doctors to better cure the patients, as it will have access to an unbiased ML algorithm to perform segmentations.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search