A short tutorial on using Autograd in Python

Contents

1 What Autograd is

Autograd is a module which computes derivatives of a given function defined in Python. This is useful when computing derivatives by hand and coding the expressions becomes rather messy.
Finding the derivatives are especially useful when writing a program which finds minimum of a cost function. Minimizing a cost function is often essential when developing programs for machine learning.
The functions to minimize can be implemented using basic mechanisms such as `for` and `while` loops, `if`-tests and recursion. In addition, Autograd supports arrays and most of the functions from `Numpy` and some functions from `Scipy`.

It is also possible to visit `the GitHub repository` for a description of Autograd, examples and some parts of the source code.

1.1 Installing Autograd

The installation of Autograd could be done in several ways, preferably using a package manager to ensure that the correct paths are set up.

1.1.1 Using pip

Autograd can be installed through `pip` by typing the following command line:

```
pip install autograd
```

2 Basic usage

Finding the gradient of any function can be done using few lines of code with Autograd. The definition of the gradient can be found in Appendix A on page 6. This section presents some basic usage of Autograd - how to set up the functions to differentiate and how to use Autograd to differentiate the functions.

2.1 Importing Numpy

As most of the problems solved using Autograd might be defined as vectors or matrices, Numpy could be useful to have available. However, due to the implemented structure in Autograd, one must import Numpy from Autograd. However, the functions from Autograd's Numpy can still be used in the same matter as you are used to from the original Numpy module.

The module to import is named `autograd.numpy`. One possible way of importing this module, is by:

```
import autograd.numpy as np
```

2.2 Defining the functions to differentiate

Now we can define a function for Autograd to differentiate. The function must be defined in the same matter as any other Python functions:

```
def a_function_name(< variables, could be more than one >):
    < Function body. You are free to use typical Python
    mechanisms such as if-tests, loops and even recursion! >
    return < function value >
```

For instance, if one wanted to differentiate the function $f(x) = x^3$, one could define it as:

```
def f(x):
    return x**3 + 1
```

It is possible to differentiate a function with more variables, for instance $f(x_1, x_2) = 3x_1^3 + x_2(x_1 - 5) + 1$

```
def f(x1,x2):
    return 3*x1**3 + x2*(x1 - 5) + 1
```

... and even more variables such as $f(x_1, x_2, x_3, x_4, x_5) = 2x_1 + 3x_2 + 5x_3 + 7x_4 + 11x_5$. Having that many variables, it might be easier to define an array representing the variables

```
def f(x):
    return 2*x[0] + 3*x[1] + 5*x[2] + 7*x[3] + 11*x[4]**2
```

Other mathematical functions are supported, such as square root, exponential and trigonometric functions. They can be found in Autograd's Numpy:

```
def f(x):
    return np.sqrt(1+x**2) + np.exp(x) + np.sin(2*np.pi*x)
```

There are many other functions from Numpy which are supported. At `page 49 in` one can find a table of which functions that are supported per 2016, and from looking at the functions in some part of the `source code` it is possible to see all of the Numpy functions that are supported at latest release.

It is possible to define functions using if-tests

```
def f(x):
    if x >= 0:
        return 1
    else:
        return 0
```

and for- and while loops

```
def f_for(x):
    val = 0
    for i in range(10):
        val = val + x**i
    return val

def f_while(x):
    val = 0
    k = 0
    while k < 10:
        val = val + x**k
        k = k + 1
    return val
```

and recursion

```
def f(n):
    if n == 1 or n == 0:
        return 1
    else:
        return n*f(n-1)
```

... and combinations of them.

2.2.1 What to be careful of

Although Autograd has a lot of features and supports a lot of functions from Numpy, extra care must be taken in some cases.

Autograd cannot[1]:

- **Allow assignment of values to arrays a function is being differentiated with respect to.** For example, if a Python function is being differentiated with respect to an array `x`, trying to set `x` at some index `i` a value `y` will produce an error

```
x[i] = y
```

- **Use the syntax `a.dot(b)` for finding the dot product.** In Numpy there are at least two ways of finding the dot product between two arrays `a` and `b`, which gives the same results and looks very similar syntax-wise. The two ways are `a.dot(b)` and `np.dot(a,b)`. The first method, `a.dot(b)`, is not supported in Autograd. Use the latter method instead:

```
np.dot(a,b)
```

Also, it is recommended to avoid using in-place operations such as `a += b` and in-place reassignment of arrays such as `u[i] = v` not being differentiated with respect to.

This possibly covers the basic pitfalls one might encounter using Autograd. However, the section "Supported and unsupported parts of numpy" in the `official tutorial` has a more detailed overview of what to avoid using Autograd.

2.3 Finding the gradient

Now that the functions have been set up, it is possible to let Autograd differentiate the functions.

To compute the gradient of a Python function, the function must be passed as an argument to Autograd's function `grad`, which can be found in the module `autograd`:

```
# Remember that grad lies in the autograd module:
from autograd import grad

# Python definition of the function
# your_function_name

# Differentiate your Python function by sending it as an argument to grad:
grad_function = grad(your_function_name)
```

The variable `grad_function` is now a function pointer to a function which represents the gradient of the given function. It is then possible to pass a value to the function pointer to evaluate the gradient at the given value:

```
grad_function = grad(your_function_name)

# Find the value of grad_function at 1:
print("The gradient of function at 1 is: ",grad_function(1.0) )
```

Note that `1.0` is passed to `grad_function`, and not `1`. Autograd cannot compute the derivative of a function when the argument is an integer, so every numeric argument most be float.

If you desire to evaluate the derivatives element wise on a returned array from your Python function, then `elementwise_grad` can be used instead:

[1] As for the time this note was written.

```
# Remember to import elementwise_grad which lies in the autograd module:
from autograd import elementwise_grad

# Python definition of the function
# your_function_name

# Let x be an array, for instance by using linspace from Numpy:
x = np.linspace(0,1,1000)

grad_function = elementwise_grad(your_function_name)

print("The derivative at all of the values in x:", grad_function(x))
```

This might come handy if you, for instance, want to plot the computed derivative of your Python function.

It might also happen that a function takes in an array and returns an array. It might be useful to find the derivative w.r.t all of the variables represented by the input array on all of the elements in the returned array. To do so, the `jacobian`[2] can be used instead:

```
# Remember to import jacobian which lies in the autograd module:
from autograd import jacobian

# Python definition of the function
# your_function_name

jacobian_function = jacobian(your_function_name)
```

It is also possible to differentiate your function as many times as you desire. As an example, the fourth derivative can be computed as:

```
grad( grad( grad( grad(your_function_name) ) ) )
```

This holds for the function `jacobian` as well.

3 Examples

Here some examples will be provided to show how Autograd can be used. The aim of these examples is to show only thr basic usage of Autograd. More examples can also be found at `its GitHub repository`.

The examples are written as Jupyter Notebooks to make experimentation and modifications of the programs easier.

3.1 Supported functions to differentiate

Section 2.2 on page 2 presented some functions which are supported to differentiate in Autograd. Since it is not a complete list, an example program has been created such that one can experiment with the supported and unsupported implementations in Autograd.

The program can be downloaded HERE. It is also demonstrated in the program how to estimate the gradient of a given Python function.

3.2 Visually comparing the derivative of a function with one variable and its analytic derivative

This example find the derivative of $f(x) = \sin\left(2\pi x + x^2\right)$. Just for comparison, the analytical derivative, $f'(x) = \cos\left(2\pi x + x^2\right) \cdot (2\pi + 2x)$ has been plotted against Autograd's computed derivative. It might be useful to plot the

[2] To see why, see Appendix B on page 7.

computed derivative to see how the growth of a function is.

The program can be downloaded HERE.

3.3 Gradient descent

3.4 Simple neural network

4 Similar libraries in other languages

There exists libraries for other programming languages that uses `automatic differentiation` to compute derivatives of a function.

A table can be found in [1, p. 24]. In the table, one can find implementations for C, C++, Fortran, MATLAB and more modules for automatic differentiation in Python.

5 Further reading

For those interested, some sources are presented below which goes deeper into Autograd and the general concept of automatic differentiation.

- The README at `the official GitHub repository`.
- A `thesis` describing Autograd from one of its developers, Dougal Maclaurin.

References

[1] "Automatic Differentiation in Machine Learning: a Survey". In: (2018). URL: `https://arxiv.org/pdf/1502.05767.pdf`.

[2] Dougal Maclaurin. "PhD". In: (2016). URL: `https://dougalmaclaurin.com/phd-thesis.pdf`.

[3] Dougal Maclaurin et al. *GitHub repository to Autograd*. URL: `https://github.com/HIPS/autograd`.

[4] Paul Vicol. *Autograd tutorial*. 2017. URL: `http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/tutorials/tut4.pdf`.

Appendices

A The gradient

The gradient of a function f is a generalization of the derivative. Suppose that the function f is defined by n variables. This can be written as $f(x_1, x_2, \ldots, x_n)$. Then its gradient, $\nabla f(x_1, x_2, \ldots, x_n)$, is a vector with the derivative of f with respect to each of the variable[3]:

$$\nabla f(x_1, x_2, \ldots, x_n) = \left(\frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_1}, \frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_2}, \ldots, \frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_n} \right)$$

For $n = 1$, that is a function with only one variable, the gradient of f is just a scalar and is the derivative of the function with respect to the variable.

[3] The derivative of f with respect to x_i is denoted at $\frac{\partial f}{\partial x_i}$

B The Jacobian

The Jacobian of a function is a further generalization of the gradient. Suppose that you have a function, in this case a *vector field*, $\vec{f}$ defined by n variables and returns a vector of m elements:

$$\vec{f}(x_1, x_2, \ldots, x_n) = (y_1,\ y_2,\ \ldots\ ,\ y_m)$$

Then the Jacobian of $\vec{f}$ is a $n \times m$ matrix, and could be thought as a gradient applied to each out the elements returned from the function:

$$\begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_{n-1}} & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_{n-1}} & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & & \vdots & \vdots \\ \frac{\partial y_{m-1}}{\partial x_1} & \frac{\partial y_{m-1}}{\partial x_2} & \cdots & \frac{\partial y_{m-1}}{\partial x_{n-1}} & \frac{\partial y_{m-1}}{\partial x_n} \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_{n-1}} & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$