Comparison of two gradient computation methods in Python

Sri Hari Krishna Narayanan
Argonne National Laboratory
Lemont, IL 60439
snarayan@anl.gov

Paul Hovland
Argonne National Laboratory
Lemont, IL 60439
hovland@anl.gov

Kshitij Kulshrestha
Paderborn University
Paderborn, Germany
kshitij@math.upb.de

Devashri Nagarkar
University of Illinois Urbana-Champaign
Champaign, IL 61801

Kaitlyn MacIntyre
Northwestern University
Evanston, IL 60208

Riley Wagner
University of Pennsylvania
Philadelphia, PA 19104

Deqing Fu
University of Chicago
Chicago, IL 60637

Gradient based optimization and machine learning applications require the computation of derivatives. For example, artificial neural networks (ANNs), a widely used learning system, use *backpropagation* to calculate the error contribution of each neuron after a batch of data is processed. Languages such as Python and R are are popular for machine learning. Therefore, there has been interest in the last few years to build tools for Python and R to compute derivatives.

Automatic Differentiation (AD), also known as autodiff and algorithmic differentiation, is a method to compute the derivatives of subprograms [9, 4]. The *adjoint* mode of AD is equivalent to backpropagation in machine learning. There have existed several tools, such as pycppad [3], pyadolc [12, 13, 11], and CasADI [2], that are Python wrappers for C++ tools . The `ad` package [15], algopy [16], and theano [10] were initial efforts to provide a native Python AD capability. Recently, autograd has emerged as a popular option to compute derivatives in Python.

Inspired in part by pyadolc's hand written interface to the AD tool ADOL-C, we have recently developed automatically generated, maintainable interfaces between ADOL-C and Python and R [7]. Here we present a performance comparison between autograd and ADOL-C and comment on the capabilities provided by the two tools.

1 Autograd and ADOL-C

Autograd differentiates native Python and Numpy code and handle a large subset of Python's features [17]. First, within a driver, the function being differentiated is supplied to autograd's `grad` function (for scalar valued output) that then returns a function that computes the original's derivative. To compute the gradient, autograd records the list of operations used to compute the outputs from the inputs. After the function is evaluated, autograd has a list of all operations that were performed and which nodes they depended on. This is the computational graph of the function evaluation. To compute the derivative, autograd applies the rules of differentiation to each node in the graph. Figure 1 shows an example of autograd usage.

ADOL-C is an operator overloading AD tool that provides accurate first- and higher-order derivatives for applications in C++ [14]. It has been used to differentiate small- and large-scale applications. ADOL-C's differentiation process consists of two steps. First, the top level function to be differentiated is *traced*. That is, ADOL-C forms a list, known as *tape*, of the operations performed by the

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

```
# Assume the computation of interest
# ay = some_function(adarray)

#Invoke grad
gradfunction = autograd.grad(some_function)


# Compute the gradient
g = gradfunction(adarray)
```

Figure 1: Autograd usage.

```
adolc.trace_on(1)
adarray = adolc.as_adouble(array)
for item in adarray.flat:
  item.declareIndependent()

# Perform main computation that computes
# ay = some_function(adarray)

ay.declareDependent()
adolc.trace_off()

g = adolc.gradient(1, adarray)
```

Figure 2: ADOL-C usage.

function for a set of input values. Driver routines can then be used to compute the original function, gradient, Jacobian, Hessian, and higher order derivatives. Figure 2 shows an example of ADOL-C usage.

2 Performance Analysis

We conducted a preliminary performance comparison of ADOL-C and autograd. We used both tools to differentiate several functions that produce scalar valued output. We fixed the size of input arrays to be 100,000 and report the average time for 10 runs. For ADOL-C we report the average time taken to trace the function as well as compute the gradient. For autograd, we report the average time taken to invoke `grad` and invoke the function returned by it. We ran the tests on machines with a Supermicro X8DTU mainboard, 2x Intel E5520 Xeon CPUs, 24GB DDR3 1066MHz Memory (6x4GB modules) with 1Gbit ethernet connected.

We emphasize that the values of the gradients generated by both tools match. Table 1 shows the performance of ADOL-C computed gradients against autograd computed gradients. The results are surprising because we see that for some tests ADOL-C is two orders of magnitude faster than autograd. For others, autograd is one order of magnitude faster than ADOL-C. The performance of the two tools certainly warrants further investigation.

3 Feature Comparison

Functions that should be differentiated in a nonstandard manner are handled elegantly by autograd using the `@primitive` decorator which allows the user to provide a custom function that computes the derivative. The ADOL-C interface to Python has no equivalent capability at the moment although, ADOL-C has the capability while differentiating C++ code [6].

For the most part, autograd does not require changes to the input code other than those noted in [17]. For ADOL-C, a few changes may be required in the computation routines. Consider the lines of code in Figure 3(a). Any mathematical functions called, such as `math.cos`, must be rewritten using the equivalent ADOL-C functions, in this case `adolc.cos`. However, if a function is performed on

Table 1: Performance comparison of ADOL-C and Autograd

Name	ADOL-C (s)		Autograd (s)
	`gradient`	`trace`	`gradient`
ackley [5]	1.20	3.744	884.41
cam [8]	0.26	2.14	846.61
correlation coefficient population	3.94	10.08	5174.16
deceptive [5]	0.51	4.34	886.01
electricpotential [18]	0.52	4.34	899.98
enthalpy [1]	0.56	4.26	0.016
exponential [5]	0.35	3.87	899.98
f6schaffer [5]	2.41	9.75	1026.81
f7schaffer [5]	4.41	10.01	908.29
langerman [5]	3.78	8.89	842.41
lunacek [5]	2.58	6.83	876.80
magneticflux	1.27	9.78	0.045
moddoublesum [5]	1.03	3.60	883.89
polygon [5]	1.19	5.93	0.037
schwefel221 [5]	0.02	1.80	858.75
schwefel222 [5]	0.52	2.35	854.09
standarddeviationpop	1.00	3.24	845.68
standarddeviationsample	1.02	3.21	846.83
torque [19]	1.07	7.21	0.032
total energy	1.19	7.65	0.029

```
secondSum = numpy.sum(numpy.cos(2.0*math.pi*array[:len(array)]))
secondSum += math.cos(2.0*math.pi*chromosome[c])
```

(a)

```
secondSum = numpy.sum(numpy.cos(2.0*math.pi*adarray[:len(adarray)]))
secondSum += adolc.cos(2.0*math.pi*adchromosome[c])
```

(b)

Figure 3: (a) Original Python computation. (b) Python computation with ADOL-C.

the whole array using `numpy`, such as `numpy.cos`, the code remains unchanged, because `numpy` can automatically call the overloaded functions from `adolc`.

Often the Jacobian or Hessian is sparse and presents the opportunity to greatly reduce storage and computational requirements in the automatically generated derivative computation. ADOL-C exploits sparsity by interfacing with ColPack, a tool that compresses structurally independent columns of the Jacobian and Hessian matrices through graph coloring approaches. To the best of the authors' knowledge autograd has no comparable capability.

4 Conclusion and Outlook

The utility of derivative based AD tools lies is their ease of use, mathematical features, and performance. We have performed a comparison two tools, ADOL-C and Python for differentiating Python and NumPy code. ADOL-C is continuously developed and the SWIG interface allows its capabilities to be used from R and Python. Autograd is still under active development with planned support for GPU operations and in-place array operations and assignment to arrays. We expect that continuous development of autograd and ADOL-C to result in a convergence of their capabilities in the long term.

Acknowledgments

We thank Michel Schanen and Daniel Maldonado for their suggestions. This work was funded in part by a grant from U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] http://www.chemteam.info/Thermochem/HessLawIntro2.html.

[2] casadi. https://github.com/casadi/casadi/wiki.

[3] pycppad. https://github.com/b45ch1/pycppad.

[4] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[5] N. Holtschulte and M. Moses. Should every man be an island. In *GECCO*. 2013.

[6] K. Kulshreshtha, S. H. K. Narayanan, and T. Albring. *A Mixed Approach to Adjoint Computation with Algorithmic Differentiation*, pages 331–340. Springer International Publishing, Cham, 2016.

[7] K. Kulshreshtha, S. H. K. Narayanan, J. Bessac, and K. MacIntyre. Efficient computation of derivatives for solving optimization problems in R and Python using SWIG-generated interfaces to ADOL-C. *Optimization Methods and Software*, 2017. Under Submission.

[8] R. S. M. Anitescu. A sparse superlinearly convergent sqp with applications to two-dimensional shape optimization. Technical report, Argonne National Laboratory, 1998.

[9] U. Naumann. *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics, 2011.

[10] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[11] S. F. Walter. PyADOLC. https://github.com/b45ch1/pyadolc.

[12] S. F. Walter. AD in Python with Application in Science and Engineering. in Eighth EuroAD Workshop, The Numerical Algorithms Group, Oxford, UK, July 2009.

[13] S. F. Walter. Algorithmic Differentiation in Python with PYADOLC and PYCPPAD. in EuroScipy Conference, Leipzig, Germany, July 2009.

[14] A. Walther and A. Griewank. Getting started with ADOL-C. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, pages 181–202. Chapman-Hall, 2012.

[15] ad 1.2.3. https://pypi.python.org/pypi/ad/1.2.3, 2014.

[16] AlgoPy, Algorithmic Differentiation in Python. https://pythonhosted.org/algopy/, 2014.

[17] Autograd. https://github.com/HIPS/autograd, 2017.

[18] Electric potential. http://physics.bu.edu//~duffy/PY106/Potential.html, 2017.

[19] Right hand rule for torque. http://hyperphysics.phy-astr.gsu.edu/hbase/tord.html, 2017.