sparsegrad Documentation

Marek Zdzislaw Szymanski

Jun 29, 2019

Contents:

CHAPTER 1

Introduction

`sparsegrad` is a Python library for automatic calculation of sparse Jacobian matrices. It is applicable to unmodified Python calculations of arbitrary complexity expressed using *supported operations*.

Assume that a Python function `f` is defined, performing some calculations with `numpy`

```
>>> import numpy as np
>>> x = np.linspace(0,1,5)
>>> def f(x):
...     return np.sqrt(x**2+1)
>>> print(f(x))
[ 1.          1.03077641  1.11803399  1.25        1.41421356]
```

Sparse Jacobian is calculated by evaluating function `f` on a suitable *seed* object:

```
>>> from sparsegrad import forward
>>> y = f(forward.seed(x))
```

Result `y` now contains sparse Jacobian information:

```
>>> print(y.dvalue)
(0, 0)        0.0
(1, 1)        0.242535625036
(2, 2)        0.4472135955
(3, 3)        0.6
(4, 4)        0.707106781187
```

`sparsegrad` is primarily intended to be used in Newton's method for solving systems of nonlinear equations. Systems with more than one million degrees of freedom per node are feasible. The algorithm of differentiation is selected and optimized to limit the runtime and the memory requirements. In some *cases*, `sparsegrad` outperforms other libraries.

`sparsegrad` is implemented in pure Python without extension modules. This simplifies both the installation and the deployment. The implementation depends only on `numpy` and `scipy` packages.

`sparsegrad` is distributed under GNU Affero General Public License version 3. The full text of the license is provided in file `LICENSE` in the root directory of distribution.

CHAPTER 2

Installation

2.1 Requirements

- Python 2.7 or Python 3.4+
- numpy >= 1.10
- scipy >= 0.14.0
- packaging >= 14.0

2.2 Installation from PyPI

This is the preferred way of installing `sparsegrad`.

Two variants of the installation are possible:

- system wide installation:

```
$ pip install sparsegrad
```

- local installation not requiring administrator's rights:

```
$ pip install sparsegrad --user
```

In the case of local installation, `sparsegrad` is installed inside user's home directory. In Linux, this defaults to `$HOME/.local`.

After installing, it is advised to run the test suite to ensure that `sparsegrad` works correctly on your system:

```
>>> import sparsegrad
>>> sparsegrad.test()
Running unit tests for sparsegrad...
```

(continues on next page)

(continued from previous page)

```
OK
<nose.result.TextTestResult run=676 errors=0 failures=0>
```

If any errors are found, `sparsegrad` is not compatible with your system. Either your Python scientific stack is too old, or there is a bug.

sparsegrad is evolving, and backward compatibility is not yet offered. It is recommended to check which version you are using:

```
>>> import sparsegrad
>>> sparsegrad.version
'0.0.6'
```

2.3 Development installation (advanced)

Current development version of sparsegrad can be installed from the development repository by running

```
$ git clone https://github.com/mzszym/sparsegrad.git
$ cd sparsegrad
$ pip install -e .
```

The option `-e` tells that `sparsegrad` code should be loaded from `git` controlled directory, instead of being copied to the Python libraries directory. As with the regular installation, `--user` option should be appended for local installation.

CHAPTER 3

Supported operations

3.1 Arithmetics

- Python scalars
- numpy `ndarray` with dimensionality <2
- broadcasting
- mathematical operators (+, -, . . .)
- numpy elementwise mathematical functions (`sin`, `exp`, . . .)

3.2 Indexing

`sparsegrad` has full support for indexing for reading arrays:

- indexing by scalars, for example `x[0]` and `x[-1]`
- indexing by slice, for example `x[::-1]`
- indexing by arrays, for example `x[np.arange(10)]`

Setting individual elements in arrays should be replaced with summing sparse vectors.

3.3 dtype promotion

`sparsegrad` does not assume a specific `dtype`. It follows `numpy` dtype coercion rules.

3.4 Branching and control flow

Since `sparsegrad` does not reuse data between evaluations, arbitrary branching of execution is allowed through Python control flow statements such as `if`. `sparsegrad` objects implements all the comparison operators.

Branching at vector element level is supported through functions `where` and `branch`.

`where` is an equivalent of the standard `numpy` function, but it supports correctly `sparsegrad` objects. As the standard version, it has a disadvantage that both possible values must be evaluated for each element. In the case of expensive calculations, this is avoided by using `branch` function, which only evaluates used values.

3.5 Sparse vectors

`sparsegrad` provides functions for summing sparse vectors with derivative information.

3.6 Irregular memory access

Collecting values from non-sequential locations in memory, with optional summing, is supported through multiplication by sparse matrix (`dot`).

Writing values to non-sequential locations in memory, with optional summing, is supported through summing sparse vectors (`sparsesum`).

3.7 Calculation of sparsity pattern

Sparsity pattern is calculating using `seed_sparsity`.

3.8 Other functions

sparsegrad provides variants of standard functions that work for both numpy and sparsegrad values:

- `dot(A,x)` : matrix - vector multiplication, where matrix A is constant
- `sum(x)` : sum of elements of a vector x
- `hstack(vecs)`, `stack(*vecs)` : concatenation of vectors *vecs*

CHAPTER 4

Architecture

`sparsegrad` performs forward mode automatic differentiation on vector valued function.

sparsegrad forward value is a pair $(\mathbf{y}, \partial\mathbf{y}/\partial\mathbf{x})$ where y is referred to as `value` and the Jacobian $\partial\mathbf{y}/\partial\mathbf{x}$ is referred to as `dvalue`.

During the evaluation, `sparsegrad` values are propagated in place of standard `numpy` arrays. This simple approach gives good result when only first order derivative is required. Most importantly, it does not involve solving graph coloring problem on the whole computation graph. For large problems, storing complete computation graph is very expensive.

When a mathematical function $\mathbf{f} = \mathbf{f}\left(\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_n}\right)$ is evaluated, the derivatives are propagated using the chain rule

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \sum_i \frac{\partial \mathbf{f}}{\partial \mathbf{y_i}} \frac{\partial \mathbf{y_i}}{\partial \mathbf{x}}$$

4.1 Support for `numpy`

To support `numpy` functions, broadcasting must be included. In the discussion below, scalars are treated as one element vectors.

Application of a numpy function involves implicit broadcasting from vector $\mathbf{y_i}$, with its proper shape, to vector $\bar{\mathbf{y}}_\mathbf{i}$, with shape of $\mathbf{f}$ if both shapes are not the same. This can be denoted by multiplication by matrix $\mathbf{B_i}$. The result of function evaluation is then

$$\mathbf{f} = \mathbf{f}\left(\mathbf{B_1 y_1}, \mathbf{B_2 y_2}, \ldots \mathbf{B_n y_n}\right)$$

and the derivative is

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \sum_i \frac{\partial \mathbf{f}}{\partial \bar{\mathbf{y}}_\mathbf{i}} \mathbf{B_i} \frac{\partial \mathbf{y_i}}{\partial \mathbf{x}}$$

The Jacobian of `numpy` function application is a diagonal matrix. If $\mathbf{g_i}$ is a `numpy` elementwise derivative of $\mathbf{f}$ with respect of to $\mathbf{y_i}$, then

$$\frac{\partial \mathbf{f}}{\partial \bar{\mathbf{y}}_\mathbf{i}} = \operatorname{diag}\left(\mathbf{g_i}\left(\mathbf{B_1 y_1}, \mathbf{B_2 y_2}, \ldots \mathbf{B_n y_n}\right)\right)$$

4.2 Optimizations

As a major optimization, the Jacobian matrices are stored as a product of scalar s, diagonal matrix $\mathrm{diag}\,(\mathbf{d})$ and a general sparse matrix $\mathbf{M}$:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = s \cdot \mathrm{diag}\,(\mathbf{d})\,\mathbf{M}$$

The general parts $\mathbf{M}$ are constant shared objects. If not specified, the diagonal parts and the general parts are assumed to be identity.

The factored matrix is referred to as `sdcsr` in `sparsegrad` code. It allows to peform the most common operations with rebuilding the general matrix part:

$$[s_1 \cdot \mathrm{diag}\,(\mathbf{d_1})\,\mathbf{M}] + [s_2 \cdot \mathrm{diag}\,(\mathbf{d_2})\,\mathbf{M}] = \mathrm{diag}\,(s_1 \cdot \mathbf{d_1} + s_2 \cdot \mathbf{d_2})\,\mathbf{M}$$

$$\alpha \cdot \mathrm{diag}\,(\mathbf{x})\,[s \cdot \mathrm{diag}\,(\mathbf{d})\,\mathbf{M}] = (\alpha s) \cdot \mathrm{diag}\,(\mathbf{x} \circ \mathbf{d})\,\mathbf{M}$$

with $\circ$ denoting elementwise multiplication.

4.3 Backward mode

Backward mode is currently not implemented because of prohibitive memory requirements for large calculations. In backward mode, each intermediate value has to accessed twice: during the forward evaluation of function value, and during backward evaluation of derivative. The memory requirements to store intermediate values are prohibitive for functions with large number of outputs, and grows linearly with the number of steps in computation.

CHAPTER 5

Runtime comparison

`sparsegrad` comes with an example of a fully implicit solver for shallow water equations. This serves as an example of a problem resulting from discretisation of partial differential equations, with realistic complexity and size. The code resides in `examples/shallow-water.ipynb`.

Runtime comparison of `sparsegrad` with ADOL-C is below. The test was run on a single core of Xeon E5-2620v4:

Calculation	ms
numpy	2.3
sparsegrad	70
ADOL-C repeated	142
ADOL-C full	2130

`numpy` only calculates function value. `sparsegrad` calculation calculates function value and derivative.

`ADOL-C repeated` calculates the function value and the derivative using computation graph and graph coloring previously stored in memory. Whole calculation is run by C code. `ADOL-C` repeated is only available when there is no change of control flow in the calculation.

`ADOL-C full` builds the computation graph, solves the graph coloring problem in addition to computing the actual output. It must be used when control flow changes in the computation leading to change in sparsity structure.

On this particular example, `sparsegrad` is from 2 to 30 times faster than `ADOL-C`.

CHAPTER 6

Tutorial

```
import sparsegrad
```

6.1 Testing installation

```
sparsegrad.test()
```

```
Running unit tests for sparsegrad
NumPy version 1.13.3
NumPy relaxed strides checking option: True
NumPy is installed in /usr/lib/python3.6/site-packages/numpy
Python version 3.6.4 (default, Dec 23 2017, 19:07:07) [GCC 7.2.1 20171128]
nose version 1.3.7
```

```
.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪..
----------------------------------------------------------------------
Ran 676 tests in 1.340s

OK
```

```
<nose.result.TextTestResult run=676 errors=0 failures=0>
```

6.2 Calculation of Jacobian

```
import numpy as np
import sparsegrad as ad
```

```
def function(x):
    return np.exp(-x**2)
```

```
x0=np.linspace(-1,1,5)
```

Calculate value and function gradient by forward mode automatic differentiation:

```
y=function(ad.forward.seed_sparse_gradient(x0))
```

Access function value:

```
y.value
```

```
array([ 0.36787944,  0.77880078,  1.        ,  0.77880078,  0.36787944])
```

Access gradient as sparse matrix:

```
y.dvalue.tocsr()
```

```
<5x5 sparse matrix of type '<class 'numpy.float64'>'
    with 5 stored elements in Compressed Sparse Row format>
```

```
print(y.dvalue.toarray())
```

```
[[ 0.73575888  0.          0.          0.          0.        ]
 [ 0.          0.77880078  0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.         -0.77880078  0.        ]
 [ 0.          0.          0.          0.         -0.73575888]]
```

6.3 Calculation of sparsity pattern

```
y=function(ad.forward.seed_sparsity(np.zeros_like(x0)))
```

Access positions of possible nonzeros in AIJ format:

```
y.sparsity.indices
```

```
array([0, 1, 2, 3, 4], dtype=int32)
```

```
y.sparsity.indptr
```

```
array([0, 1, 2, 3, 4, 5], dtype=int32)
```

Access positions of possible nonzeros as scipy CSR matrix:

```
y.sparsity.tocsr()
```

```
<5x5 sparse matrix of type '<class 'numpy.int64'>'
    with 5 stored elements in Compressed Sparse Row format>
```

```
print(y.sparsity.tocsr().toarray())
```

```
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

CHAPTER 7

Reference

7.1 sparsegrad package

7.1.1 Subpackages

sparsegrad.base package

Submodules

sparsegrad.base.expr module

class sparsegrad.base.expr.**expr_base**
Bases: object

Base class for numpy-compatible operator overloading

It provides default overloads of arithmetic operators and methods for mathematical functions. The default overloads call abstract apply method to calculate the result of operation.

absolute(***kwargs*)

classmethod apply(*func*, *args*)
Apply DifferentiableFunction to args

apply1(*func*)
Apply single argument DifferentiableFunction to value

arccos(***kwargs*)

arccosh(***kwargs*)

arcsin(***kwargs*)

arcsinh(***kwargs*)

arctan(***kwargs*)

arctanh(***kwargs*)

compare(*operator*, *other*)

cos(***kwargs*)

cosh(***kwargs*)

exp(***kwargs*)

expm1(***kwargs*)

log(***kwargs*)

log1p(***kwargs*)

negative(***kwargs*)

reciprocal(***kwargs*)

sign(***kwargs*)

sin(***kwargs*)

sinh(***kwargs*)

sqrt(***kwargs*)

square(***kwargs*)

tan(***kwargs*)

tanh(***kwargs*)

Module contents

Base for operator overloading

sparsegrad.forward package

Submodules

sparsegrad.forward.forward module

sparsegrad.forward.forward.**value**
alias of sparsegrad.forward.forward.forward_value

sparsegrad.forward.forward.**seed**(*x*, *T=<class 'sparsegrad.forward.forward.forward_value'>*)

sparsegrad.forward.forward.**seed_sparse_gradient**(*x*, *T=<class 'sparsegrad.forward.forward.forward_value'>*)

sparsegrad.forward.forward.**seed_sparsity**(*x*, *T=<class 'sparsegrad.forward.forward.forward_value_sparsity'>*)

sparsegrad.forward.forward.**nvalue**(*x*)
return numeric value of x, x of type (forward_value, numeric types)

Module contents

Forward mode automatic differentiation

sparsegrad.impl package

Subpackages

sparsegrad.impl.sparse package

Submodules

sparsegrad.impl.sparse.sparse module

This module contains implementation details sparse matrix operations

class sparsegrad.impl.sparse.sparse.**sdcsr**(*mshape*, *s=array(1)*, *diag=array(1)*, *M=None*)
Bases: object

Scaled matrix, which is stored as

$$s \cdot diag(\mathbf{diag}) \cdot \mathbf{M}$$

where s is scalar, diag is row scaling vector (scalar and vector allowed), and M is general part (None is allowed to indicate diagonal matrix).

mshape stores matrix shape. None for mshape[0] denotes differentation of scalar. None for mshape[1] denotes differentiation with repsect to scalar.

No copies of M, diag are made, therefore they must be constant objects.

broadcast(*output*)
Return broadcast matrix $\mathbf{B_{output}}$ for broadcasting x to output, this matrix being Jacobian of x

chain(*output*, *x*)
Apply chain rule for elementwise operation

Jacobian of elementwise operation is $diag(\mathbf{x})$. Return $diag(\mathbf{B_{output}} \cdot \mathbf{x}) \cdot \mathbf{B_{output}} \cdot \mathbf{self}$

classmethod fma(*output*, **terms*)
Apply chain rule to elementwise functions

Returns sum(d.chain(output,x) for x,d in terms)

classmethod fma2(*output*, **terms*)
Apply chain rule to elementwise functions

Returns sum(d.chain(output,x) for x,d in terms)

getitem_arrayp(*output*, *idx*)
Generate Jacobian matrix for operation output=x[idx], this matrix being Jacobian of x. idx is array with all entries positive.

getitem_general(*output*, *idx*)
Generate Jacobian matrix for operation output=x[idx], this matrix being Jacobian of x. General version.

classmethod new(*mshape*, *diag=array(1)*, *M=None*)
Alternative constructor, which checks dimension of diag and assigns to scalar/vector part properly

rdot(*y*, *other*)
Return Jacobian of $\mathbf{y} = \mathbf{other} \cdot \mathbf{self}$, with $\cdot$ denoting matrix multiplication.

sum()
Return Jacobian of y=sum(x), this matrix being Jacobian of x

tovalue()
Return this matrix as standard CSR matrix. The result is cached.

vstack(*output*, *parts*)
Return Jacobian of output=hstack(parts)

zero(*output*)
Return empty Jacobian, which would result from output=0*x, this matrix being Jacobian of x.

class sparsegrad.impl.sparse.sparse.**sparsity_csr**(*mshape*, *s=None*, *diag=None*, *M=None*)
Bases: *sparsegrad.impl.sparse.sparse.sdcsr*

This is a variant of matrix only propagating sparsity information

chain(*output*, *x*)
Apply chain rule for elementwise operation

Jacobian of elementwise operation is $diag(\mathbf{x})$. Return $diag(\mathbf{B_{output}} \cdot \mathbf{x}) \cdot \mathbf{B_{output}} \cdot \mathbf{self}$

classmethod fma(*output*, **terms*)
Apply chain rule to elementwise functions

Returns sum(d.chain(output,x) for x,d in terms)

classmethod fma2(*output*, **terms*)
Apply chain rule to elementwise functions

Returns sum(d.chain(output,x) for x,d in terms)

rdot(*y*, *other*)
Return Jacobian of $\mathbf{y} = \mathbf{other} \cdot \mathbf{self}$, with $\cdot$ denoting matrix multiplication.

sparsegrad.impl.sparse.sparse.**sample_csr_rows**(*csr*, *rows*)
return (indptr,ix) such that csr[rows]=csr_matrix((csr.data[ix],csr.indices[ix],indptr))

sparsegrad.impl.sparse.sparse.**csr_matrix**
alias of sparsegrad.impl.sparse.sparse.csr_matrix_nochecking

sparsegrad.impl.sparse.sparse.**csc_matrix**
alias of sparsegrad.impl.sparse.sparse.csc_matrix_unchecked

Module contents

sparsegrad.impl.sparsevec package

Submodules

sparsegrad.impl.sparsevec.sparsevec module

This module contains implementation details of summing sparse vectors.

sparsegrad.impl.sparsevec.sparsevec.**sparsesum**(*terms*, *hstack=<function hstack>*, *nvalue=<function <lambda>>*, *wrap=<function <lambda>>*, *check_unique=False*, *return_sparse=False*)
Sum sparse vectors

This is a general function, which propagates index information and numerical values. Suitable functions must be supplied as arguments to propagate other information.

terms [list of sparsevec] terms

hstack [callable(vectors)] function to use for concatenating vectors

nvalue [callable(vector)] function to use for extracting numerical value

wrap [callable(idx, v, result)] function to use for wrapping the result, with idx, v being concatenated inputs

check_unique [bool] whether to perform test for double assignments (useful when this function is used to replace item assignment)

return_sparse [bool] whether to calculate sparse results

class sparsegrad.impl.sparsevec.sparsevec.**sparsevec**(*n*, *idx*, *v*)
Bases: object

Sparse vector of length n, with nonzero entries in arrays (idx,v) where idx contains the indices of entries, and v contains the corresponding values

Module contents

Module contents

This module can be imported before everything else and used to redirect some of scipy functionality. Rest of sparsegrad uses scipy functions imported here.

sparsegrad.impl.**dot_**(*a*, *b*)
Proxy for a.dot(b)

sparsegrad.sparsevec package

Submodules

sparsegrad.sparsevec.sparsevec module

sparsegrad.sparsevec.sparsevec.**sparsesum**(*terms*, ***kwargs*)
Generalized version of sparsesum

Module contents

7.1.2 Submodules

7.1.3 sparsegrad.func module

7.1.4 sparsegrad.utils module

7.1.5 Module contents

Sparse Jacobian calculation of numpy expressions

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

S

T

V

Z