Lazyarray Documentation

Release 0.6.0

Andrew P. Davison, Joël Chavas, Elodie Legouée (CNRS) and An

Oct 20, 2024

CONTENTS

lazyarray is a Python package that provides a lazily-evaluated numerical array class, `larray`, based on and compatible with NumPy arrays.

Lazy evaluation means that any operations on the array (potentially including array construction) are not performed immediately, but are delayed until evaluation is specifically requested. Evaluation of only parts of the array is also possible.

Use of an `larray` can potentially save considerable computation time and memory in cases where:

- arrays are used conditionally (i.e. there are cases in which the array is never used);
- only parts of an array are used (for example in distributed computation, in which each MPI node operates on a subset of the elements of the array).

It appears that much of this functionality may appear in a future version of NumPy (see this discussion on the numpy-discussion mailing list), but at the time of writing I couldn't find anything equivalent out there. DistNumPy might also be an alternative for some of the use cases of lazyarray.

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Dependencies

- Python >= 3.8
- numpy >= 1.20
- (optional) scipy >= 1.7

1.1.2 Installing from the Python Package Index

```
$ pip install lazyarray
```

This will automatically download and install the latest release (you may need to have administrator privileges on the machine you are installing on).

1.1.3 Installing from source

To install the latest version of lazyarray from the Git repository:

```
$ git clone https://github.com/NeuralEnsemble/lazyarray
$ cd lazyarray
$ pip install .
```

1.2 Tutorial

The `lazyarray` module contains a single class, `larray`.

```
>>> from lazyarray import larray
```

1.2.1 Creating a lazy array

Lazy arrays may be created from single numbers, from sequences (lists, NumPy arrays), from iterators, from generators, or from a certain class of functions. Here are some examples:

```
>>> from_number = larray(20.0)
>>> from_list = larray([0, 1, 1, 2, 3, 5, 8])
>>> import numpy as np
>>> from_array = larray(np.arange(6).reshape((2, 3)))
>>> from_iter = larray(iter(range(8)))
>>> from_gen = larray((x**2 + 2*x + 3 for x in range(5)))
```

To create a lazy array from a function or other callable, the function must accept one or more integers as arguments (depending on the dimensionality of the array) and return a single number.

```
>>> def f(i, j):
...     return i*np.sin(np.pi*j/100)
>>> from_func = larray(f)
```

Specifying array shape

Where the `larray` is created from something that does not already have a known shape (i.e. from something that is not a list or array), it is possible to specify the shape of the array at the time of construction:

```
>>> from_func2 = larray(lambda i: 2*i, shape=(6,))
>>> print(from_func2.shape)
(6,)
```

For sequences, the shape is introspected:

```
>>> from_list.shape
(7,)
>>> from_array.shape
(2, 3)
```

Otherwise, the `shape` attribute is set to `None`, and must be set later before the array can be evaluated.

```
>>> print(from_number.shape)
None
>>> print(from_iter.shape)
None
>>> print(from_gen.shape)
None
>>> print(from_func.shape)
None
```

1.2.2 Evaluating a lazy array

The simplest way to evaluate a lazy array is with the `evaluate()` method, which returns a NumPy array:

```
>>> from_list.evaluate()
array([0, 1, 1, 2, 3, 5, 8])
>>> from_array.evaluate()
array([[0, 1, 2],
       [3, 4, 5]])
>>> from_number.evaluate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/andrew/dev/lazyarray/lazyarray.py", line 35, in wrapped_meth
    raise ValueError("Shape of larray not specified")
ValueError: Shape of larray not specified
>>> from_number.shape = (2, 2)
>>> from_number.evaluate()
array([[ 20.,  20.],
       [ 20.,  20.]])
```

Note that an `larray` can only be evaluated once its shape has been defined. Note also that a lazy array created from a single number evaluates to a homogeneous array containing that number. To obtain just the value, use the `simplify` argument:

```
>>> from_number.evaluate(simplify=True)
20.0
```

Evaluating a lazy array created from an iterator or generator fills the array in row-first order. The number of values generated by the iterator must fit within the array shape:

```
>>> from_iter.shape = (2, 4)
>>> from_iter.evaluate()
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]])
>>> from_gen.shape = (5,)
>>> from_gen.evaluate()
array([  3.,   6.,  11.,  18.,  27.])
```

If it doesn't, an Exception is raised:

```
>>> from_iter.shape = (7,)
>>> from_iter.evaluate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    from_iter.evaluate()
  File "/Users/andrew/dev/lazyarray/lazyarray.py", line 36, in wrapped_meth
    return meth(self, *args, **kwargs)
  File "/Users/andrew/dev/lazyarray/lazyarray.py", line 235, in evaluate
    x = x.reshape(self.shape)
ValueError: total size of new array must be unchanged
```

When evaluating a lazy array created from a callable, the function is called with the indices of each element of the array:

```
>>> from_func.shape = (3, 4)
>>> from_func.evaluate()
array([[ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.03141076,  0.06279052,  0.09410831],
       [ 0.        ,  0.06282152,  0.12558104,  0.18821663]])
```

It is also possible to evaluate only parts of an array. This is explained below.

1.2.3 Performing operations on a lazy array

Just as with a normal NumPy array, it is possible to perform elementwise arithmetic operations:

```
>>> a = from_list + 2
>>> b = 2*a
>>> print(type(b))
<class 'lazyarray.larray'>
```

However, these operations are not carried out immediately, rather they are queued up to be carried out later, which can lead to large time and memory savings if the evaluation step turns out later not to be needed, or if only part of the array needs to be evaluated.

```
>>> b.evaluate()
array([ 4,  6,  6,  8, 10, 14, 20])
```

Some more examples:

```
>>> a = 1.0/(from_list + 1)
>>> a.evaluate()
array([ 1.        ,  0.5       ,  0.5       ,  0.33333333,  0.25      ,
        0.16666667,  0.11111111])
>>> (from_list < 2).evaluate()
array([ True,  True,  True, False, False, False, False], dtype=bool)
>>> (from_list**2).evaluate()
array([ 0,  1,  1,  4,  9, 25, 64])
>>> x = from_list
>>> (x**2 - 2*x + 5).evaluate()
array([ 5,  4,  4,  5,  8, 20, 53])
```

Numpy ufuncs cannot be used directly with lazy arrays, as NumPy does not know what to do with `larray` objects. The lazyarray module therefore provides lazy array-compatible versions of a subset of the NumPy ufuncs, e.g.:

```
>>> from lazyarray import sqrt
>>> sqrt(from_list).evaluate()
array([ 0.        ,  1.        ,  1.        ,  1.41421356,  1.73205081,
        2.23606798,  2.82842712])
```

For any other function that operates on a NumPy array, it can be applied to a lazy array using the `apply()` method:

```
>>> def g(x):
...     return x**2 - 2*x + 5
>>> from_list.apply(g)
>>> from_list.evaluate()
array([ 5,  4,  4,  5,  8, 20, 53])
```

1.2.4 Partial evaluation

When accessing a single element of an array, only that element is evaluated, where possible, not the whole array:

```
>>> x = larray(lambda i,j: 2*i + 3*j, shape=(4, 5))
>>> x[3, 2]
12
>>> y = larray(lambda i: i*(2-i), shape=(6,))
>>> y[4]
-8
```

The same is true for accessing individual rows or columns:

```
>>> x[1]
array([ 2,  5,  8, 11, 14])
>>> x[:, 4]
array([12, 14, 16, 18])
>>> x[:, (0, 4)]
array([[ 0, 12],
       [ 2, 14],
       [ 4, 16],
       [ 6, 18]])
```

1.2.5 Creating lazy arrays from SciPy sparse matrices

Lazy arrays may also be created from SciPy sparse matrices. There are 7 different sparse matrices.

- csc_matrix(arg1[, shape, dtype, copy]) Compressed Sparse Column matrix
- csr_matrix(arg1[, shape, dtype, copy]) Compressed Sparse Row matrix
- bsr_matrix(arg1[, shape, dtype, copy, blocksize]) Block Sparse Row matrix
- lil_matrix(arg1[, shape, dtype, copy]) Row-based linked list sparse matrix
- dok_matrix(arg1[, shape, dtype, copy]) Dictionary Of Keys based sparse matrix.
- coo_matrix(arg1[, shape, dtype, copy]) A sparse matrix in COOrdinate format.
- dia_matrix(arg1[, shape, dtype, copy]) Sparse matrix with DIAgonal storage

Here are some examples to use them.

Creating sparse matrices

Sparse matrices comes from SciPy package for numerical data. First to use them it is necessary to import libraries.

```
>>> import numpy as np
>>> from lazyarray import larray
>>> from scipy.sparse import bsr_matrix, coo_matrix, csc_matrix, csr_matrix, dia_
↪matrix, dok_matrix, lil_matrix
```

Creating a sparse matrix requires filling each row and column with data. For example :

```
>>> row = np.array([0, 2, 2, 0, 1, 2])
>>> col = np.array([0, 0, 1, 2, 2, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
```

The 7 sparse matrices are not defined in the same way.

The bsr_matrix, coo_matrix, csc_matrix and csr_matrix are defined as follows :

```
>>> sparr = bsr_matrix((data, (row, col)), shape=(3, 3))
>>> sparr = coo_matrix((data, (row, col)), shape=(3, 3))
>>> sparr = csc_matrix((data, (row, col)), shape=(3, 3))
>>> sparr = csr_matrix((data, (row, col)), shape=(3, 3))
```

In regards to the dia_matrix :

```
>>> data_dia = np.array([[1, 2, 3, 4]]).repeat(3, axis=0)
>>> offsets = np.array([0, -1, 2])
>>> sparr = dia_matrix((data_dia, offsets), shape=(4, 4))
```

For the dok_matrix :

```
>>> sparr = dok_matrix(((row, col)), shape=(3, 3))
```

For the lil_matrix :

```
>>> sparr = lil_matrix(data, shape=(3, 3))
```

In the continuation of this tutorial, the sparse matrix used will be called sparr and refers to the csc_matrix.

It is possible to convert the sparse matrix as a NumPy array, as follows:

```
>>> print(sparr.toarray())
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])
```

Specifying the shape and the type of a sparse matrix

To know the shape and the type of the sparse matrices, you can use :

```
>>> larr = larray(sparr)
>>> print (larr.shape)
(3, 3)
>>> print (larr.dtype)
dtype('int64')
```

Evaluating a sparse matrix

Evaluating a sparse matrix refers to the evaluate() method, which returns a NumPy array :

```
>>> print (larr.evaluate())
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])
```

When creating a sparse matrix, some values may remain empty. In this case, the evaluate () method has the argument, called empty_val, referring to the special value nan, for Not a Number, defined in NumPy. This method fills these empty with this nan value.

```
>>> print (larr.evaluate(empty_val=np.nan))
array([[1, nan, 4],
       [nan, nan, 5],
       [2, 3, 6]])
```

Accessing individual rows or columns of a sparse matrix

To access specific elements of the matrix, like individual rows or columns :

```
>>> larr[2, :]
```

In this case, the third line of the sparse matrix is obtained. However, this method is different depending on the sparse matrices used :

For csc_matrix and csr_matrix :

```
>>> print (larr[2, :])
array([2, 3, 6])
```

During execution, the matrices bsr_matrix, coo_matrix and dia_matrix, do not support indexing. The solution is to convert them to another format. It is therefore necessary to go through csr_matrix in order to perform the calculation.

```
>>> print(sparr.tocsr()[2,:])
```

Depending on the definition given previously to the matrix, for the dok_matrix :

```
>>> print (larr[1, :])
```

And for lil_matrix :

```
>>> print (larr[0, :])
```

In case we want to access an element of a column, we must proceed in the same way as previously, by changing index. Here is an example of how to access an item in the third column of the sparse matrix.

```
>>> larr[:, 2]
```

Finally, to have information on the sparse matrix :

```
>>>print (larr.base_value)
<3x3 sparse matrix of type '<class 'numpy.int64'>'
    with 6 stored elements in Compressed Sparse Column format>
```

1.3 Performance

The main aim of lazyarray is to improve performance (increased speed and reduced memory use) in two scenarios:

- arrays are used conditionally (i.e. there are cases in which the array is never used);
- only parts of an array are used (for example in distributed computation, in which each MPI node operates on a subset of the elements of the array).

However, at the same time use of `larray` objects should not be too much slower than plain NumPy arrays in normal use.

Here we see that using a lazyarray adds minimal overhead compared to using a plain array:

```
>>> from timeit import repeat
>>> repeat('np.fromfunction(lambda i,j: i*i + 2*i*j + 3, (5000, 5000))',
...        setup='import numpy as np', number=1, repeat=5)
[1.9397640228271484, 1.92628812789917, 1.8796701431274414, 1.6766629219055176, 1.
↪6844701766967773]
>>> repeat('larray(lambda i,j: i*i + 2*i*j + 3, (5000, 5000)).evaluate()',
...        setup='from lazyarray import larray', number=1, repeat=5)
[1.686661958694458, 1.6836578845977783, 1.6853220462799072, 1.6538069248199463, 1.
↪645576000213623]
```

while if we only need to evaluate part of the array (perhaps because the other parts are being evaluated on other nodes), there is a major gain from using a lazy array.

```
>>> repeat('np.fromfunction(lambda i,j: i*i + 2*i*j + 3, (5000, 5000))[:, 0:4999:10]',
...        setup='import numpy as np', number=1, repeat=5)
[1.691796064376831, 1.668884038925171, 1.647057056427002, 1.6792259216308594, 1.
↪652547836303711]
>>> repeat('larray(lambda i,j: i*i + 2*i*j + 3, (5000, 5000))[:, 0:4999:10]',
...        setup='from lazyarray import larray', number=1, repeat=5)
[0.23157119750976562, 0.16121792793273926, 0.1594078540802002, 0.16096210479736328, 0.
↪16096997261047363]
```

Note: These timings were done on a MacBook Pro 2.8 GHz Intel Core 2 Duo with 4 GB RAM, Python 2.7 and NumPy 1.6.1.

1.4 Reference

class lazyarray.**larray**(*value*, *shape=None*, *dtype=None*)

Optimises storage of and operations on arrays in various ways:

- stores only a single value if all the values in the array are the same;
- if the array is created from a function *f(i)* or *f(i,j)*, then elements are only evaluated when they are accessed. Any operations performed on the array are also queued up to be executed on access.

Two use cases for the latter are:

- to save memory for very large arrays by accessing them one row or column at a time: the entire array need never be in memory.
- in parallelized code, different rows or columns may be evaluated on different nodes or in different threads.

evaluate(*simplify=False*)

Return the lazy array as a real NumPy array.

If the array is homogeneous and `simplify` is `True`, return a single numerical value.

apply(*f*)

Add the function *f(x)* to the list of the operations to be performed, where *x* will be a scalar or a numpy array.

```
>>> m = larray(4, shape=(2,2))
>>> m.apply(np.sqrt)
```

(continues on next page)

(continued from previous page)

```
>>> m.evaluate()
array([[ 2.,  2.],
       [ 2.,  2.]])
```

property is_homogeneous

True if all the elements of the array are the same.

property ncols

Size of the second dimension (if it exists) of the array.

property nrows

Size of the first dimension of the array.

property shape

Shape of the array

1.5 Developers' guide

TO BE COMPLETED

1.5.1 Testing

In the *test* sub-directory, run:

```
$ pytest
```

To see how well the tests cover the code base, run:

```
$ pytest --cov=lazyarray --cov-report html --cov-report term-missing
```

1.5.2 Making a release

- Update the version numbers in pyproject.toml, lazyarray.py, doc/conf.py and doc/installation.txt
- Update changelog.txt
- Run all the tests with Python 3
- python -m build
- twine upload dist/lazyarray-<version>*
- Commit the changes, tag with release number, push to Github
- Rebuild the documentation at http://lazyarray.readthedocs.org/

CHAPTER

TWO

LICENCE

The code is released under the Modified BSD licence.

INDEX