fastats Documentation

Release a1

Dave Willmer

Dec 30, 2022

Contents:

CHAPTER 1

Single Page Docs

Everything you need on one page :)

1.1 Introduction

Fastats is a numerical library for python designed to give the highest level of performance in terms of both development time and execution time.

It does this by providing a high-level, easy-to-use python API which is specifically designed to be JIT-compiled using the excellent numba library to run at native speed.

Here's a quick example to show some basic concepts:

```
import numpy as np

from fastats import single_pass

data = np.random.random((1000000,))


def square_minus_one(x):
    return x * x - 1.0


result = single_pass(data, value=square_minus_one)
```

`fastats.core.single_pass()` is a core fastats function which takes a numpy array and a function, and applies the function to each row of the numpy array. The function must be passed as the keyword argument `value`.

The `value` function can be any user-defined or library python function which is able to be JIT-compiled in `nopython` mode by numba.

Note: **Technical note:** Internally, fastats takes the `value` function and replaces all occurrences of that function recursively using the AST, then JIT-compiles the resulting function using numba so that everything runs at native

speed. This ability to pass a function as an argument to a numba function is one of the key tenets of the fastats library. This is achieved using the `@fs` decorator from the fastats library, which will be described later.

This is conceptually similar to the `pandas.apply` or numpy `ufunc` system for vectorizing calculations across arrays - the benefits of fastats are (1) performance and (2) no required knowledge of decorators, C/Cython or ufuncs.

Fastats provides:

- A mechanism for passing arbitrary functions to numba functions, and having them JIT correctly.
- An AST transform which can cleanly replace functions within a nested python hierarchy.
- A core library of iteration functions to allow users to avoid explicit indexing in most cases. This is not only safer but also usually faster to develop and faster to execute.
- A core library of high-performance linear algebra functions which are generally much faster than existing implementations.

The AST transform in combination with the linear algebra routines allow us to perform many standard data science/time series analysis tasks much faster than with existing packages. The idea of using numba as a JIT compiler is not just for immediate performance benefits, but also so that these routines are always compiled for the hardware they are running on - this is one of the main reasons that fastats can be faster than existing pre-compiled libraries.

With fastats, functions can be wrapped with the `@fs` decorator which transforms the function to allow any keyword arguments to be passed; these keyword arguments specify the functions to be replaced within the original code.

When you specify a keyword argument, the code is transformed into AST form, and is then re-built with the specified functions replaced. This leads to extremely fast execution times (due to numba), and extremely fast development times (due to AST replacement of functions).

1.1.1 Performance Comparison

Here we use the data from the introductory code block above - a numpy array of 1 million float64's. The test machine is a 2015 Macbook Pro, Intel Core i7, 16GB RAM:

System	Code	Result
Pandas .apply	import pandas as pd s = pd.Series(data) %timeit s.apply(square_minus_one)	391 ms ± 11.9 ms per loop
Numpy funcs	import numpy as np %timeit np.power(data, 2) - 1	30.8 ms ± 944 µs per loop
Pandas binops	import pandas as pd s = pd.Series(data) %timeit (s * s) - 1	4.3 ms ± 93.8 µs per loop
Numpy binops	import numpy as np %timeit (data ** 2) - 1	1.65 ms ± 19.7 µs per loop
Numba direct	from numba import jit jit_wrap = jit(nopython=True, parallel=True)(square_minus_one) %timeit jit_wrap(data)	1.6 ms ± 22.2 µs per loop
Fastats	from fastats import single_pass my_func = single_pass(data, value=square_minus_one, return_callable=True) %timeit my_func(data)	1.48 ms ± 22.4 µs per loop

As you can see, without resorting to C/Cython or other native code, fastats allows you to apply arbitrary Python functions at native speeds.

1.2 Core Functionality

This section details the main functionality available in the `fastats` library.

1.2.1 single_pass

We have already used the `single_pass` function in the introductory example above. In a simplistic sense, this is equivalent to the following Python code:

```
def single_pass(x, value=None):
    result = np.zeros_like(x)
    for row in x:
        result[i] = value(row)
    return result
```

As you can see, it allocates a new array the same size and type of the input array, then iterates over the input array row-by-row, and assigns the return value to the corresponding row in the newly allocated output array.

Note: The actual implementation is slightly more nuanced than this because it will recursively replace all calls to the `value` function (or any other keywords specified) throughout the entire subtree.

Note: This `single_pass` implementation is slightly naive in that it doesn't currently account for multiple input parameters to the *value* function. This is something we are actively working on.

To use *single_pass* you always pass it a numpy array and at least one keyword argument `value=` with the function you would like to apply:

```
def my_calc(x):
    return x**3 - x - 1

result = single_pass(data, value=my_calc)
```

In the `my_calc` example above, the function object is shown passed as the `value=` keyword argument. This function will be replaced within the existing `single_pass` definition and automatically JIT-compiled using numba. The resulting iteration will then occur at native speeds, not bottlenecked by python bytecode interpretation.

Note: The `fastats` ethos of allowing extremely fast development as well as execution time implies that we need to help stop developers/data scientists writing their own iteration code. It's far safer to use a pre-built and well-tested function for iteration, especially where explicit indexing is used. `single_pass` facilitates this as it requires no knowledge of any indexing issues, and is also faster than most/all other methods - it is therefore the recommended way to apply any operation to an array.

1.2.2 windowed_pass

Another common operation is to apply a function as a rolling window along a data set; for this we have `windowed_pass`. Here's a slightly contrived example showing how to calculate the rolling mean across an array:

```
from fastats import windowed_pass

def mean(x):
    return np.sum(x) / x.size

result = windowed_pass(x, 10, value=mean)
```

In the code above, the obvious difference between `single_pass` and `windowed_pass` is that the windowed version takes a second positional argument to specify the size of the window. A window of that size is then fed into the `value=` function, so the function also needs to accept an array, not a scalar value.

To change the window from 10 items to 25 items, we would therefore change line 6 above to:

```
result = windowed_pass(x, 25, value=mean)
```

For a more real-world example, let's take a look at a rolling least-squares regression (OLS) across an array. The OLS code will be discussed in the `linear_algebra` section later, but for now let's look at how to perform a high-performance rolling OLS over a numpy array with 2 columns:

```
from fastats import windowed_pass
from fastats.linear_algebra import ols

def ols_wrap(x):
    return ols(x[:,:1], x[:,1:])[0][0]

result = windowed_pass(x, 10, value=ols_wrap)
```

The `ols_wrap` function is due to a current limitation of `fastats` - you cannot currently pass a function requiring more than one argument to `windowed_pass`; the workaround is to write a one-line wrap function which passes each column of the data set correctly to the underlying function, as shown above. This limitation will be removed in a future version.

Note: Similar to the `single_pass` function, this `windowed_pass` example will take the high-performance `ols` implementation from the `fastats` standard library, and JIT compile both the OLS and the loop which performs the iteration. In this way the user gets high performance without needing to know the underlying details of performance coding.

This is a quick way to get a rolling OLS function, however usually we are not interested in the slope values - most people will be more interested in the `r^2` and `t-statistic` values. To do this we will need to return multiple values from the wrap function:

```
from fastats import windowed_pass
from fastats.linear_algebra import ols, r_squared

def ols_r_squared(x):
    out = np.zeros(2)
    a = x[:, :1]
    b = x[:, 1:]
    slope = ols(a, b)[0][0]
    r2 = r_squared(a, b)[0][0]
    out[0] = slope
    out[1] = r2
    return out

result = windowed_pass(data, 10, value=ols_r_squared)
```

Note: This wrap function for multiple return values is a bit fiddly to set up, but we are actively working on making this easier.

This works because the return value from the `windowed_pass` function is always the same shape as the input data. As a result with the raw input data in two columns, we can support two return values from the inner jitted function. This is however quite restrictive, and we will improve this in a future release.

To call a numpy function as a rolling window, you currently need to wrap the numpy call in another function:

```
from fastats import windowed_pass

def nanmean(x):
    return np.nanmean(x)

result = windowed_pass(x, 10, value=nanmean)
```

We will work to remove this restriction in a future release.

1.2.3 windowed_stateful_pass

1.2.4 Early JIT compilation without execution - *return_callable*

Most users will find it useful to store the jitted function, rather than re-jitting the code every time it is called. To do this, pass the keyword argument `return_callable=True` to any core function (a `core` function is one with an `@fs` decorator) and it will return the function instead of executing it.

For example:

```
rolling_ols = windowed_pass(x, 10, value=ols_wrap, return_callable=True)
```

After this, you can call the `rolling_ols` function without incurring another AST transform and JIT compilation cycle. The result is just a normal Python function, so you call it with its expected arguments:

```
result = rolling_ols(x, 10)
large_window_result = rolling_ols(x, 250)
```

1.2.5 The *@fs* decorator

The ability to pass functions as arguments to numba functions and have everything jitted correctly comes from the @fs decorator. When this decorator is applied to a function, it changes the semantics of keyword arguments; keyword arguments indicate that any function calls of that argument name should be replaced with the function being passed in. For example:

```
def mean(x):
    return np.sum(x) / x.size

result = single_pass(x, value=mean)
```

The definition of `single_pass` does not take any keyword arguments, or specify `**kwargs`, but it has an `@fs` decorator and therefore substitutes all occurences of `value` for `mean` in its own function body, and in the function bodies of any of its child functions.

1.3 fastats.linear_algebra

1.4 fastats.maths

1.5 fastats.optimise

1.6 Technical Details

1.6.1 How do the AST Transforms work?

Imagine a nested function such as a very simple Newton-Raphson solver:

```
def newton_raphson(x0, delta, root=fs_func, deriv=deriv):
    last_x = x0
    next_x = last_x + 10 * delta
    while abs(last_x - next_x) > delta:
        new_y = root(next_x)
        last_x = next_x
        next_x = last_x - new_y / deriv(last_x, delta)
    return next_x
```

In the code above we actually want to replace the `root` function with one of our own choosing, without having to re-write the entire newton_raphson wrapper function.

In `fastats` this is performed by changing the semantics of positional and keyword arguments; `numba` does not allow us to pass functions as arguments, but even if it did, we still need the ability to arbitrarily modify deeply nested functions (use cases discussed below), rather than just the calls in the top-level function.

The Newton-Raphson code shown above is in the `fastats` standard library, and allows us to do the following:

```
from fastats.optimise.root_finding.newton_raphson import newton_raphson

def my_func(x):
    return x ** 3 - x - 1

result = newton_raphson(0.5, 0.001, root=my_func)
assert 1.324 < result < 1.325
```

`my_func` is the function for which we would like to find the roots. It takes one argument `x`, which will be varied to find the root.

When `newton_raphson` is called, it takes the *root=my_func* kwarg, and inspects the signature of `my_func`. It finds that `my_func` takes 1 argument and expects the first argument (`x` in this case) to be the parameter that is modified by the algorithm to find the root.

This is what `fastats` performs: at any level in the AST, `fastats` will modify the function signatures and ensure that the correct arguments are passed to all functions, in order to allow any function to be modified by passing it as a keyword argument at the top-level.

`fastats` does not currently support multiple arguments to solver functions, but we will support this in a future release.

In this example, `deriv` will numerically calculate the derivative at each point, which is the reason for requiring nested function substitution from the AST transform. It is also possible to use the same `deriv` function for a wide range of `root` functions, as the numerical derivative just calls the `root` function with the x-values bumped in either direction.

As a result, we need the `root=` keyword argument to replace all values in the child functions (including `deriv`), not just the one at the top-level. This allows us to have a very efficient and easy-to-learn API without sacrificing performance.

However, in this case we can trivially calculate the analytical derivative by hand, and replace the `deriv` function like this:

```
def my_deriv(x, y):
    return 2 * x ** 2 - 1

result = newton_raphson(0.5, 0.001, root=my_func, deriv=my_deriv)
```

Which allows us to **optionally** pass an optimised function to calculate the derivative, but fall back on an unoptimised version for fast experimenting/research.

This is not-limited to specific functions - if you are happy with lower-precision in certain calculations, you can pass faster (lower-precision) versions of any mathematical functions, and `fastats` will replace them throughout the entire AST before passing the code onto numba, without requiring you to modify any code.

As an example, some calculations require the complementary error function `erfc` to be calculated. The accuracy (precision) of `erfc` depends partially on how many terms are in the expansion. By reducing the number of terms we can speed up calculations at the expense of accuracy.

If you are happy with 8 decimal places, and you had a a custom function `erfc8` to calculate this in an optimal manner, you could speed up calculations like this:

```
my_solve = newton_raphson(0.5, 0.001, root=my_func, erfc=erfc8)
```

To increase precision (at the expense of calculation time), you could use `erfc16`:

```
my_solve = newton_raphson(0.5, 0.001, root=my_func, deriv=my_deriv, erfc=erfc16)
```

These *ast-modification* semantics therefore allow you to use any pure python numerical code which can be JIT-compiled, regardless of whether the original author allowed arbitrary functions to be passed in.

1.6.2 Why are we re-writing functions that already exist in numpy/scipy/etc?

One of the major advantages of numba is that the JIT compilation will be optimized for the specific hardware you are running on. The traditional Python system of writing a C or Cython routine and then pre-compiling it will not be optimised for specific hardware.

Over the last few years, the SIMD registers in CPUs have grown to 512-bits, which numba/fastats code will be able to take advantage of. Older builds of numpy/scipy will not.

This library therefore attempts to move as much logic into small reusable python functions which can be compiled as required using numba. We do not use pre-compiled code.

This system also allows us to have lower or higher precision variants of functions if we want to control the runtime and/or accuracy. With compiled code this adds a huge overhead to the codebase.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search