Here's how you can get some free speed on your Python code with Numba

Easily turn your Python code into fast machine code

George Seif
Jun 11 · 5 min read ★

"We can't use Python, it's too slow."

Anyone who's worked with Python long enough has probably heard a statement like that at one point.

The person who said it isn't wrong either. Relative to many other programming languages, Python *is slow*. Benchmark game has some solid benchmarks for comparing the speed of various programming languages on different tasks.

A common method for solving this speed problem is to re-write your code in something fast like C++ and then throw a Python wrapper on top. That'll get you the speed of C++ while maintaining the ease of using Python in your main application.

The challenge with that of course is that you'll have to re-write the code in C++; that's a pretty time consuming process.

The Python library **Numba** gives us an easy way around that challenge—free speed ups without having to write any code other than Python!

Introducing Numba

Numba is a compiler library that transforms Python code into optimised machine code. With this transformation, Numba can make numerical algorithms that are written in Python approach the speeds of C code.

You don't need to do anything fancy with your Python code either. Just add a single line before the Python function you want to optimise and Numba will do the rest!

We can install Numba using pip:

```
pip install numba
```

If your code has a lot of numerical operations, uses Numpy a lot, and/or has a lot of loops, then Numba should give you a good speedup.

Speeding up Python loops

The most basic use of Numba is in speeding up those dreaded Python for-loops.

First off, if you're using a loop in your Python code, it's always a good idea to first check if you can replace it with a numpy function. There are of course, cases where numpy doesn't have the function you want.

In our first example, we're going to write a function for the Insertion sorting algorithm in Python. The function will take an unsorted list as input and return the sorted list as an output.

The code below first constructs a list of 100,000 random integers. Then, we apply insertion sort to the list 50 times in a row and measure the average speed across all 50 of the sort operations.

100,000 numbers is quite a lot of numbers to be sorting, especially when our sorting algorithm has an average complexity of O(n^2). On my PC with an i7–8700k, sorting all of those numbers takes an average of 3.0104 seconds!

```python
import time
import random

num_loops = 50
len_of_list = 100000

def insertion_sort(arr):

    for i in range(len(arr)):
        cursor = arr[i]
        pos = i

        while pos > 0 and arr[pos - 1] > cursor:
            # Swap the number down the list
            arr[pos] = arr[pos - 1]
            pos = pos - 1
        # Break and do the final swap
        arr[pos] = cursor

    return arr

start = time.time()
list_of_numbers = list()
```

Python loops are known to be slow. It's even worse here in our example where we have a while-loop *inside* of a for-loop. Plus, since our sorting algorithm is O(n²), as we add more items to our list, our run-time increases quadratically!

Let's speed things up with numba.

When we see a function that contains a loop written in pure Python, it's usually a good sign that numba can help. Check out the code below to see how it works.

```python
from numba import jit
import time
import random

num_loops = 50
len_of_list = 100000

@jit(nopython=True)
def insertion_sort(arr):

    for i in range(len(arr)):
        cursor = arr[i]
        pos = i

        while pos > 0 and arr[pos - 1] > cursor:
            # Swap the number down the list
            arr[pos] = arr[pos - 1]
            pos = pos - 1
        # Break and do the final swap
        arr[pos] = cursor

    return arr

start = time.time()
```

We've only added 2 extra lines to our code. The first one is an import statement to import the **jit** decorator. The second is our usage of the jit decorator on our function.

Applying the jit decorator to our function signals to numba that we want to apply the transformation to machine code to the function.

The **nopython** argument specifies if we want Numba to use purely machine code or to fill in some Python code if necessary. This should usually be set to true to get the best performance unless you find that Numba throws an error.

That's all there is to it! Just add the **@jit(nopython=True)** above your function and Numba will take care of the rest!

On my PC, , sorting all of those numbers takes an average of 0.1424 seconds—that's a 21X speed up!

```python
from numba import jit
import time
import random

num_loops = 50
len_of_list = 100000

@jit(nopython=True)
def insertion_sort(arr):

    for i in range(len(arr)):
        cursor = arr[i]
        pos = i

        while pos > 0 and arr[pos - 1] > cursor:
            # Swap the number down the list
            arr[pos] = arr[pos - 1]
            pos = pos - 1
        # Break and do the final swap
        arr[pos] = cursor

    return arr

start = time.time()
```

Speeding up Numpy operations

Another area where Numba shines is in speeding up operations done with Numpy. This time, we're going to add together 3 fairly large arrays, about the size of a typical image, and then square them using the `numpy.square()` function.

Check out the code below to see how that works in Python with a bit of Numpy.

```
import time

import numpy as np

num_loops = 50
img_1 = np.ones((1000, 1000), np.int64) * 5
img_2 = np.ones((1000, 1000), np.int64) * 10
img_3 = np.ones((1000, 1000), np.int64) * 15

def add_arrays(img_1, img_2, img_3):
    return np.square(img_1 + img_2 + img_3)

start = time.time()

```

Notice that whenever we do basic array computations on Numpy arrays such as adding, multiplying, and squaring, the code is automatically vectorised internally by Numpy. That's why it's often much better for performance to replace pure Python code with Numpy wherever possible.

The above code has an average run time of 0.002288 seconds for combining the arrays on my PC.

But even Numpy code isn't as fast as the machine optimised code that Numba goes down to. The code below will perform the exact same array operations as before. This time, we've added the **vectorize** decorator just above the function, signalling to numba that it should perform the machine code transformation on our function.

```python
from numba import vectorize, int64
import time

import numpy as np

num_loops = 50
img_1 = np.ones((1000, 1000), np.int64) * 5
img_2 = np.ones((1000, 1000), np.int64) * 10
img_3 = np.ones((1000, 1000), np.int64) * 15

@vectorize([int64(int64, int64, int64)], target="paral
def add_arrays(img_1, img_2, img_3):
    return np.square(img_1 + img_2 + img_3)

start = time.time()
```

The vectorize decorator takes two inputs.

The first specifies the input type of the numpy array you will be operating on. This must be specified since Numba uses it to transform the code into its most optimal version. By knowing the input type before hand, Numba will be able to figure out exactly how your array can be most efficiently stored and operated on.

The second input is called the "target". It specifies *how* you would like to run your function:

- **cpu:** for running on a single CPU thread
- **parallel:** for running on a multi-core, multi-threaded CPU
- **cuda:** for running on the GPU

The *parallel* option tends to be much faster than the *cpu* option in almost all cases. The *cuda* option is mainly useful for very large arrays with many parallelizable operations, since in that case we can fully utilise the advantage of having so many cores on the GPU.

The above code has an average run time of 0.001196 seconds for combining the arrays on my PC—that's about a 2X speedup. Not bad for adding a single line of code!

Is it always super fast?

Numba is going to be most effective when applied in either of these areas:

- Places where Python code is slower than C code (typically loops)
- Places where the same operation is applied to an area (i.e the same operation on many elements)

Outside of those areas, Numba probably won't be giving you much speed. Since the advantage that comes with converting to the lower level code is gone in that case.

In general, it's always worth giving it a quick try. Adding a single line of code above a few Python functions is worth a shot a speeding up your code by 2 to 21X!

• • •

Like to learn?

Follow me on twitter where I post all about the latest and greatest AI, Technology, and Science! Connect with me on LinkedIn too!

Recommended Reading

Want to learn more about coding in Python? The ***Python Crash Course*** book is the best resource out there for learning how to code in Python!

And just a heads up, I support this blog with Amazon affiliate links to great books, because sharing great books helps everyone! As an Amazon Associate I earn from qualifying purchases.