Introduction

This lesson gives a brief introduction to code vectorization and explains it with the help of an example.

Problem vectorization is much harder than code vectorization because it means that you fundamentally have to rethink your problem in order to make it vectorizable. Most of the time this means you have to use a different algorithm to solve your problem or even worse... to invent a new one. The difficulty is thus to think out-of-the-box.

To illustrate this, let's consider a simple problem where given two vectors `X` and `Y`, we want to compute the sum of `X[i]*Y[j]` for all pairs of indices `i`, `j`.

One simple and obvious solution is given below.

```python
def compute_python(X, Y):
    result = 0
    for i in range(len(X)):
        for j in range(len(Y)):
            result += X[i] * Y[j]
    return result

print(compute_python([1,2],[1,2]))
```

RUN SAVE RESET

However, this first and naïve implementation requires two loops and we already know it will be slow:

main.py
tools.py

```python
import numpy as np
from tools import timeit

def compute_python(X, Y):
    result = 0
    for i in range(len(X)):
        for j in range(len(Y)):
            result += X[i] * Y[j]
            return result

X = np.arange(1000)
timeit("compute_python(X,X)", globals())
```

RUN SAVE RESET

Output 0.771s

```
10000 loops, best of 3: 2.11 usec per loop
```

How to vectorize the problem then? If you remember your linear algebra course, you may have identified the expression `X[i] * Y[j]` to be very similar to a matrix product expression. So maybe we could benefit from some NumPy speedup. One wrong solution would be to write:

```python
def compute_numpy_wrong(X, Y):
    return (X*Y).sum()

print(compute_numpy_wrong([1,2],[1,2]))
```

This is wrong because the `X*Y` expression will actually compute a new vector `Z` such that `Z[i] = X[i] * Y[i]` and this is not what we want. Instead, we can exploit NumPy broadcasting by first reshaping the two vectors and then multiplying them:

```python
def compute_numpy(X, Y):
    Z = X.reshape(len(X),1) * Y.reshape(1,len(Y))
    return Z.sum()
```

Here we have `Z[i,j] == X[i,0]*Y[0,j]` and if we take the sum over each elements of `Z`, we get the expected result. Let's see how much speed we gain in the process:

main.py
tools.py

```python
import numpy as np
from tools import timeit
def compute_numpy(X, Y):
    Z = X.reshape(len(X),1) * Y.reshape(1,len(Y))
    return Z.sum()

X = np.arange(1000)
timeit("compute_numpy(X,X)", globals())
```

RUN SAVE RESET

Output 0.721s

```
10 loops, best of 3: 2.21 msec per loop
```

This is better, we gained a factor of ~150. But we can do much better.

If you look again and more closely at the pure Python version, you can see that the inner loop is using `X[i]` that does not depend on the `j` index, meaning it can be removed from the inner loop. Code can be rewritten as:

```python
def compute_numpy_better_1(X, Y):
    result = 0
    for i in range(len(X)):
        Ysum = 0
        for j in range(len(Y)):
            Ysum += Y[j]
        result += X[i]*Ysum
    return result
```

But since the inner loop does not depend on the `i` index, we might as well compute it only once:

```python
def compute_numpy_better_2(X, Y):
    result = 0
    Ysum = 0
    for j in range(len(Y)):
        Ysum += Y[j]
        for i in range(len(X)):
            result += X[i]*Ysum
            return result
```

Not so bad, we have removed the inner loop, transforming $O(n^2)$ complexity into $O(n)$ complexity. Using the same approach, we can now write:

```python
def compute_numpy_better_3(x, y):
    Ysum = 0
    for j in range(len(Y)):
        Ysum += Y[j]
        Xsum = 0
        for i in range(len(X)):
            Xsum += X[i]
            return Xsum*Ysum
```

Finally, having realized we only need the product of the sum over `X` and `Y` respectively, we can benefit from the `np.sum` function and write:

```python
def compute_numpy_better(x, y):
    return np.sum(y) * np.sum(x)
```

It is shorter, clearer and much, much faster:

main.py
tools.py

```python
import numpy as np
from tools import timeit
def compute_numpy_better(x, y):
    return np.sum(y) * np.sum(x)

X = np.arange(1000)
timeit("compute_numpy_better(X,X)", globals())
```

RUN SAVE RESET

Output 0.694s

```
1000 loops, best of 3: 10.5 usec per loop
```

We have indeed reformulated our problem, taking advantage of the fact that $\sum_{ij} X_i Y_j = \sum_i X_i \sum_j Y_j$ and we've learned in the meantime that there are two kinds of vectorization:

- Code vectorization
- Problem vectorization

The latter is the most difficult but the most important because this is where you can expect huge gains in speed. In this simple example, we gain a factor of 150 with code vectorization but we gained a factor of 70,000 with problem vectorization, just by writing our problem differently (even though you cannot expect such a huge speedup in all situations).

However, code vectorization remains an important factor, and if we rewrite the last solution the Python way, the improvement is good but not as much as in the NumPy version:

```python
def compute_python_better(x, y):
    return sum(x)*sum(y)
```

RUN SAVE RESET

✓ Succeeded

This new Python version is much faster than the previous Python version, but still, it is 50 times slower than the NumPy version:

main.py
tools.py

```python
import numpy as np
from tools import timeit
def compute_python_better(x, y):
    return sum(x)*sum(y)

X = np.arange(1000)
timeit("compute_python_better(X,X)", globals())
```

RUN SAVE RESET

Output 1.136s

```
100 loops, best of 3: 423 usec per loop
```

Next, we will discuss a case study and see how we can solve it using the problem vectorization approach.