Project: Image Processing Optimization

Course: Advanced Python

Prepared by: Ksenia Saenko(ks4841), Stella Sun(ss8955), James Guo (xg626)

Term: Spring 2018

1. PROBLEM DESCRIPTION

Image processing is painfully slow because we need to access each pixel, therefore, we have to include a lot of “for loops” and “if else” statement in the code. For almost every step in Image Processing and Pattern Recognition, pixel manipulation is an essential part of the entire process.

However, due to the growing dimensionationally of images, the inefficiency of traversing images in Python is becoming the bottleneck of the program. More and more default operations are implemented in machine learning libraries like scikit-learn to avoid the high time penalty of the running time. Meanwhile, except for common utility functions, sometimes we still need to write some customized functions.

Efficient image recognition is an acute problem today and can be applied to many domains, such as self-driving cars.

Fig.1: Examples of image detection

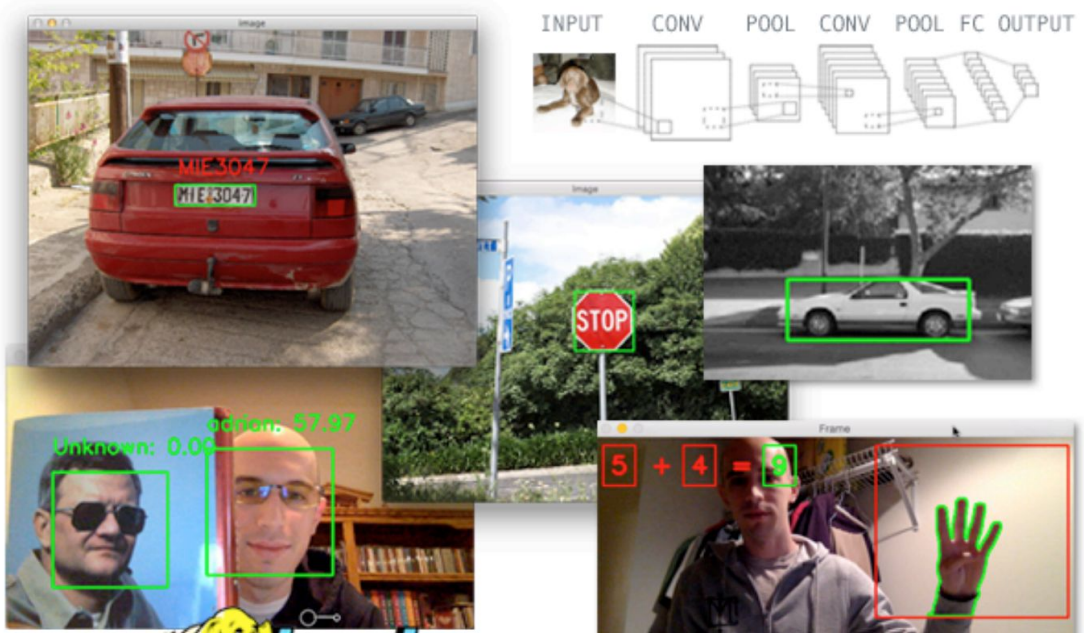


Fig.2: example traversing image with for-loops

```
# Gausssion smoothing: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm
def smooth_image_with_Gaussian_filter( img ):
        kernel = (0.006, 0.061, 0.242, 0.383, 0.242, 0.061, 0.006)
        kernel_size = len( kernel )
        border_offset = ( kernel_size - 1 ) / 2

        img_copy = np.copy( img )
        for i in range( 0, row ):
                # Keep border values as they are
                for j in range( border_offset, col - border_offset ):
                        img_copy_ij = 0
                        for k in range( (-1)*border_offset, border_offset + 1 ):
                                img_copy_ij += img[ i ][ j+k ] * kernel[ border_offset + k ]
                        img_copy[i][j] = img_copy_ij

        img_copy_copy = np.copy( img_copy )
        # Keep border values as they are
        for i in range( border_offset, row - border_offset ):
                for j in range( 0, col ):
                        img_copy_copy_ij = 0
                        for k in range( (-1)*border_offset, border_offset + 1 ):
                                img_copy_copy_ij += img_copy[ i+k ][ j ] * kernel[ border_offset + k ]
                        img_copy_copy[i][j] = img_copy_copy_ij

        return img_copy_copy
```

2. METHODOLOGY

- **Built-in libraries**: such as OpenCV, Scikit-Image
- **Vector Operation/Matrix Multiplication**: aggregate the pixel level or kernel level operations and replace them with the equivalent matrix multiplication operation, which can be further optimized in Numpy library functions. Recall what we did in homework 3 in heat diffusion.
- **Cython:** fix the data type and avoid Python type checking penalty.
- **Open-MP**(Open multi-processing): Optional
- PySpark(Parallel computing): Optional
- Implement Python performance tuning that we learned in class, such as reducing function overhead.

3. COURSE TOOLS AND METHODS WE'LL USE

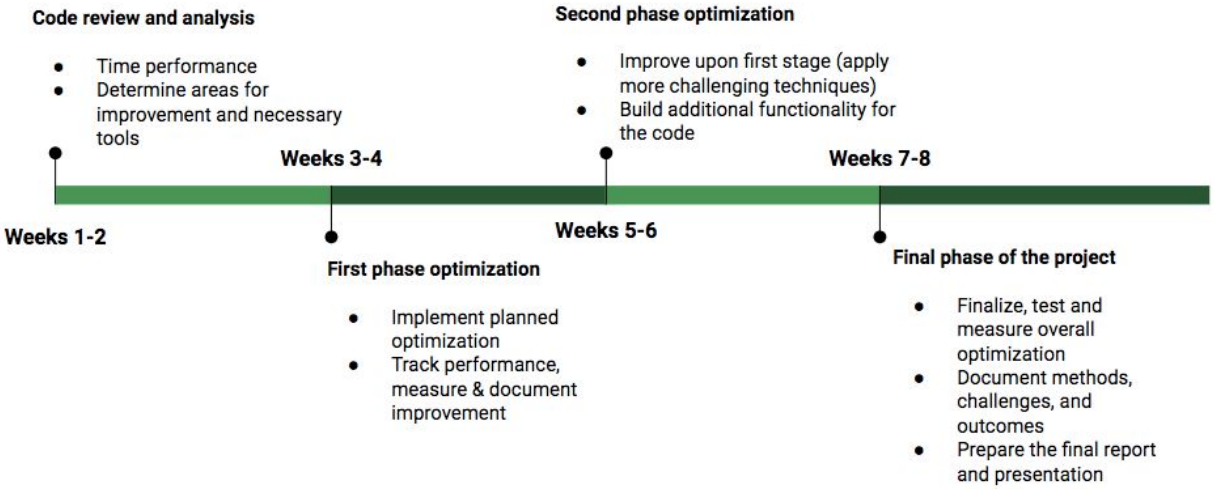
We will try to use the following tools and methods:

- Python Performance
    - Reducing function overhead
    - Efficient membership testing
    - Minimizing loops
    - Using built-in functions
- Itertools
- Numpy
- Cython
- Numba
- Timing and profiling tools
    - cProfile
    - timeit
    - Prun
- Version Control: Github repository
- Comparison with implementations in other languages, such as Go, Java, etc, if time permits.

4. EXPECTED RESULTS

We'd like to focus on the speed of the implementation instead of the readability and accuracy of the result. In other words, we will not change the code beyond the updates for improving the speed. Our goal is to decrease the time of the core parts of the code by at least 20%. We will identify the bottleneck of the code and determine which updates improved efficiency the most.

5. SCHEDULE



6. REFERENCES

Jupyter notebooks from Advanced Python class.
Fast, optimized 'for' pixel loops with OpenCV and Python