Oxford Internet Institute, University of Oxford

Assignment Cover Sheet

Candidate Number *Please note, your OSS number is NOT your candidate number*	1033256
Assignment *e.g. Online Social Networks*	Data Analytics at Scale
Term *Term assignment issued, e.g. MT or HT*	MT
Title/Question *Provide the full title, or If applicable, note the question number and the FULL question from the assigned list of questions*	A High Performance Google Images Scraper and Processing Algorithm
Word Count	4364

By placing a tick in this box ☑ I hereby certify as follows:

(a) This thesis or coursework is entirely my own work, except where acknowledgments of other sources are given. I also confirm that this coursework has not been submitted, wholly or substantially, to another examination at this or any other University or educational institution;

(b) I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at https://www.ox.ac.uk/students/academic/guidance/skills?wssl=1.

(c) I agree that my work may be checked for plagiarism using Turnitin software and have read the Notice to Candidates which can be seen at: http://www.admin.ox.ac.uk/proctors/turnitin2w.shtml and that I agree to my work being screened and used as explained in that Notice;

(d) I have clearly indicated (with appropriate references) the presence of all material I have paraphrased, quoted or used from other sources, including any diagrams, charts, tables or graphs.

(e) I have acknowledged appropriately any assistance I have received in addition to that provided by my [tutor/supervisor/adviser].

(f) I have not sought assistance from a professional agency;

(g) I understand that any false claims for this work will be reported to the Proctors and may be penalized in accordance with the University regulations.

Please remember:

- To attach a second relevant cover sheet if you have a disability such as dyslexia or dyspraxia. These are available from the Higher Degrees Office, but the Disability Advisory Service will be able to guide you.

A High Performance Google Images Scraper and Processing Algorithm

Candidate Number:1033256

January 14, 2019

1 Introduction

1.1 A note about the attached files

The goal of this assignment is to report how certain code was optimized. The attached ZIP file contains two folders, `Original/`, which contains the original version, and `Optimized/`, which contains the optimized version. The instructions to run the algorithm and install the required libraries are the same for both.

1.2 Motivation

Our activity online reflects many aspects of our culture and behavior, including stereotypes. Previous studies have shown that ordinary human language presents gender and other types of biases that are reflected in word embeddings (Caliskan, Bryson, & Narayanan 2017), a machine learning algorithm that allows to represent words as vectors and therefore to understand semantic relationships between them. Furthermore, Garg, Schiebinger, Jurafsky, & Zou (2017) studied how word embeddings trained on 100 years of text data reflect occupation and demographic changes over time.

These biases are also present in Internet images. Kay, Matuszek and Munson (2015) downloaded the first 100 result for a list of professions and hired participants to manually count the ratio of women in these results, finding that this number was related with the percentage of women in these professions reported in census data. Furthermore, they found evidence suggesting that Google Images search results overestimated the gender inequalities reported in the census (i.e. showing a smaller proportion of women than the census reported in female-dominated professions, while presenting a bigger proportion of men than the census reported in male-dominated professions).

Our research question builds on this idea and tries to estimate the proportion of women in the first 500 Google Image search results for different professions (e.g. architect, janitor, receptionist, model, etc.) and to find its correlation with the census data. However, an important difference with the aforementioned study is that, instead of using participants to estimate the proportion of women in the list of pictures resulting from each search term, we will accomplish this without human intervention. Using an algorithm for image processing grants an almost unlimited scalability that will allow to study this phenomenon for different domains without increasing the cost of the study.

1.3 The algorithm

To estimate the proportion of women in the Google Images search results for a certain search term, the algorithm needs to: (1) scrape and download the Google Images search results, (2) detect the faces in these pictures, and (3) inference their gender.

(1) **Web Scraping** (`DownloadImages.py`): Vasa (2018) developed a web scraper that allows to download all the images returned by a certain Google Images search. Additionally, we used a small excerpt of code to check if the downloaded images are in a valid format (PNG or JPG), and if not try to convert them into JPG and remove them if this is not possible. Having the desired search terms in a text file (with each search term in a line; see Professions.txt for an example), we can run this code using:

```
$ python DownloadImages.py Professions.txt.
```

This will create a new folder inside `Downlaoded_Images/` with the name of the search term and put there the downloaded images for that search term.

(2) **Face detection (`CropFaces.py`)**: To detect the faces in the pictures we used the face recognition module (Geitgey, 2018) that uses a pre-trained Histogram of Oriented Gradients and Support Vector Machine included in the `dlib` (King, 2009) library. This takes the images in each folder that was generated by `DownloadImages.py` (e.g. `Downlaoded_Images/Nurse/`) and creates a new folder inside `Cropped_faces/` (e.g. `Downlaoded_Images/Nurse/ Cropped_faces/`). Inside `Cropped_faces/` it puts pictures of the faces cropped from the Google Image search results. Additionally, it creates a text file for each face with the location of 68 facial landmarks.

```
$ python CropFaces.py Professions.txt
```

(3) **Gender classification** (DetectGender.py): The last step is to use a Convolutional Neural Network pre-trained by Levi and Hassner (2015) in order to identify the cropped faces as male or female. It was specifically developed to handle strong variations in lightning and row-solution pictures, which makes it ideal for Google Images search results. DetectGender.py takes the images in the `Cropped_faces/` folder (e.g. `Downlaoded_Images/Nurse/Cropped_faces/`) and creates a CSV inside Datasets/ with the name of the search term (e.g. `/Datasets/Nurse.csv`). The CSV has four rows: (1) the file name; (2) Probability of the face being from a female (i.e. output of the neural network); (3) height of the image; and (4) width of the image.

```
$ python DetectGender.py Professions.txt
```

It is important to highlight that these Python scripts should be run sequentially. For convenience, we have included a file named `Pipeline.py` that automatically executes these files in order. Just use:

```
$ python Pipeline.py Professions.txt
```

It could be argued that `CropFaces.py` is unnecessarily saving the cropped faces into images. It would be easier to skip this intermediate output and just use the Python representation of the picture to identify the gender instead of having to load the image again. Moreover, detecting the 68 facial landmarks does also not make much sense, because these text files are never used in the code. However, it should be explained that there is an additional Python script that computes the face average based on this information. This part was excluded from this assignment.

1.4 Data Analysis and Results

The algorithm described in the previous section put inside the `Datasets/` folder the corresponding CSV files. Additionally, we have another CSV file in the main directory which contains the same list of professions that we had in Professions.txt and the corresponding percentage of women for each profession according to the census[1].

In sum, the first step of the data analysis consisted in reading Professions.csv and adding one column containing the percentage of women in the Google Images search results. Obviously, that required reading and processing all the DetectGender.py output CSV files. Two important decisions were taken for the analysis. The first one was that not all the classified faces would be taken into account. Some professions require usual interactions with people outside that occupation (e.g. lawyers and paralegals with clients, doctors and counselors with patients), and these people tend to appear in the pictures. To avoid counting clients and patients, at least to a certain extent, we decided to just use the biggest face in each picture. The second decision was to use a cutoff of 0.5 for converting the probability of being a woman (i.e. the output of the Convolutional Neural Network) into the binary variable Man, Woman. Considering this, we calculated the proportion of women in the Google Image search results for all the professions, obtaining a correlation with the census data of r = 0.869. This supports the hypothesis that the Google Image search results reflect at some extent the percentage of women in professions. We can visualize this relationship in a scatter plot.

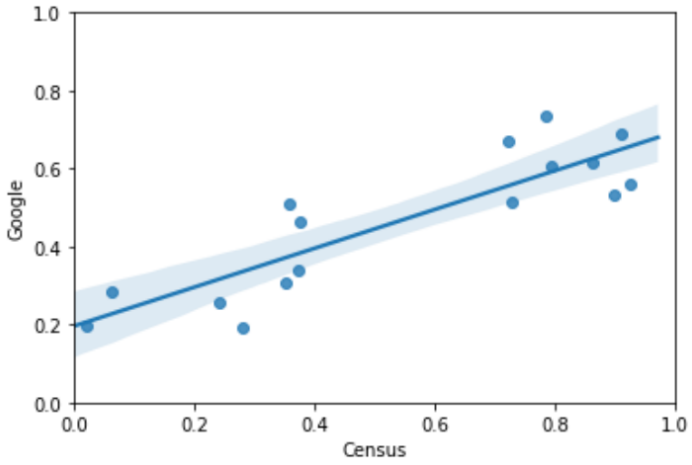

Figure 1: Relationship between the percentage of women in the Google Images search results and the official census data for 16 professions.

Note that the code for data analysis is in the `Data_Analysis.ipynb` file. A much more detailed analysis that includes 104 Professions can be seen in Candidate 1033256 (2019).

1.5 Installation and requirements

This code uses highly popular libraries and more specific modules for each of the Python scripts, which we will describe below. The packages that are not built-in and should be installed before running the code are: `numpy`, `pandas`, `SciPy`, `seaborn`, `json`, and `tqdm`. We highly recommend using Anaconda[2], a popular Python data science platform which

[1] `https://www.bls.gov/cps/cpsaat11.htm`
[2] `https://www.anaconda.com/`

includes some of these packages and allows to create environments with different packages and versions of Python, apart from some other advantages.

(1) Web Scraping

`DownloadImages.py` requires installing `Selenium`, `google_images_download`, and `PIL/Pillow`.

```
$ pip install selenium
$ pip install google_images_download
$ pip install pillow
```

Additionally, there are two pre-requisites. The first one is to have Google Chrome installed. The second is to download Chromedriver[3] and put it in the project directory (i.e. `Original/` in the case of the original code and `Optimized/` in the case of the optimized code).

(2) Face detection

CropFaces requires installing `dlib`, `face_recognition`, and `opencv-contrib-python`.

```
$ pip install dlib
$ pip install face_recognition
$ pip install opencv-contrib-python
```

(3) Gender Classification

The first step is to install `Caffe`[4]. After that, it is necessary to download the files containing the Convolutional Neural Network and that are available on the project page on Github[5]. Only the Caffe model for gender classification, the deploy prototext, and the mean image are necessary. They should be put in the `cnn_age_gender/` folder.

2 Optimization

The main goal of this assignment is to optimize the Python files described above. It is however important to take into account that there are significant limitations, mainly because the code was already written with performance in mind. The face detection and gender classification are based on `dlib` and `Caffe`, respectively. The two libraries are highly optimized and written in C++, which means that our code is already performing much better than other implementations built in Python. The statistical techniques that we are using are also quite efficient. The SVM-HOG for detecting faces has an accuracy slightly worse than CNN trained for the same task, but it is much faster. The CNN that we use for gender recognition just has three convolutional layers (96, 256, and 384 filter, respectively) and two fully connected layers (with 512 neurons each), which is relatively lightweight compared to modern standards. Additionally, note that the main bottleneck with the web scraping script is the network limitations. Even with a fast Internet connection, we cannot force the servers that contain the images to send them quicker.

Below we will comment on some approaches to optimize the code. When the optimization can be written in a few lines of code (e.g. list comprehensions) both the unoptimized and optimized version will be shown. When the optimization requires more lines of code it will be omitted (e.g. multiprocessing), although it can be checked comparing the corresponding files in the `Original/` and `Optimized/` folders.

[3] `http://chromedriver.chromium.org/`
[4] `http://caffe.berkeleyvision.org/`
[5] `https://talhassner.github.io/home/publication/2015_CVPR`

2.1 Approaching optimization

Deciding what parts of the code should be optimized is not an easy task. The first step is profiling, that is, to identify what parts are taking longer or being executed more often. Probably the easier tool for measuring the execution time of a small code snippet is `timeit`, which can also be called from a Jupyter Notebook. It is what we used to estimate the execution times for the code snippets shown in the following sections. However, this doesn't give us an in-depth understanding about the code. In this sense, the `line_profiler`[6] module allows us to see a line-by-line profiling of functions. For example, we defined a function with the contents inside the first for loop in `Original/CropFaces.py`, obtaining:

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def CropFaces(searchterm):
     2         1         93.0     93.0      0.0      imagefolder = os.path.join(os.getcwd(),"Downloaded_images",searchterm)
     3         1         41.0     41.0      0.0      targetfolder = os.path.join(imagefolder, "Cropped_faces")
     4
     5        99       3967.0     40.1      0.0      for image_path in os.listdir(imagefolder):
     6        98        296.0      3.0      0.0          try:
     7        98    5178594.0  52842.8      2.2              image = face_recognition.load_image_file(os.path.join(imagefolder, image_path))
     8        97  193637727.0 1996265.2     81.7               face_locations = face_recognition.face_locations(image)
     9        97    5161323.0  53209.5      2.2              img = cv2.imread(os.path.join(imagefolder, image_path))
    10        97       1769.0     18.2      0.0              height, width, channels = img.shape
    11
    12       198       3434.0     17.3      0.0              for index in range(0,len(face_locations)):
    13       101        596.0      5.9      0.0                  x = face_locations[index][0]-50
    14       101        422.0      4.2      0.0                  y = face_locations[index][1]+30
    15       101        357.0      3.5      0.0                  w = face_locations[index][2]+30
    16       101        338.0      3.3      0.0                  h = face_locations[index][3]-30
    17       101        296.0      2.9      0.0                  if x < 0:
    18        20         53.0      2.6      0.0                      x = 0
    19       101        279.0      2.8      0.0                  if w > height:
    20                                                               w = height
    21       101        275.0      2.7      0.0                  if h < 0:
    22         1         15.0     15.0      0.0                      h = 0
    23       101        275.0      2.7      0.0                  if y > width:
    24                                                               y = width
    25
    26       101      87726.0    868.6      0.0                  sub_face = img[x:w, h:y]
    27
    28       101       2256.0     22.3      0.0                  targetfile = targetfolder + "/" + image_path[:-4] + "_" + str(index) + ".jpg"
    29       101    1403798.0  13899.0      0.6                  cv2.imwrite(targetfile, sub_face)
    30       101       1161.0     11.5      0.0                  try:
    31       101   31043378.0 307360.2     13.1                      face_landmarks = _raw_face_landmarks(sub_face)
    32       101        972.0      9.6      0.0                      landmarks_as_tuples = [[(p.x, p.y) for p in face_landmarks.parts()]
    33       101      68993.0    683.1      0.0                                             for face_landmarks in face_landmarks]
    34       101     122157.0   1209.5      0.1                      fh = open(targetfile[:-4] + ".txt","w")
    35      6901      17330.0      2.5      0.0                      for p in landmarks_as_tuples[0]:
    36      6800      62389.0      9.2      0.0                          fh.write(str(p[0]) + " " + str(p[1]) + "\n")
    37       100     303352.0   3033.5      0.1                      fh.close()
    38         1         14.0     14.0      0.0                  except:
    39         1          5.0      5.0      0.0                      pass
    40         1          4.0      4.0      0.0          except:
    41         1          4.0      4.0      0.0              pass
```

Around 80% of the time is spent detecting the faces and 13% generating the 68 face landmarks. These two process cannot be optimized as they are being pushed to C++ by `dlib` and are already highly efficient. However, with this line-by-line profiling we can detect some inefficiencies. For example, we see that we are loading each each image one time for detecting the faces (with the name `image`) and then loading the image again for cropping it (calling it `img`). We removed this inefficiency in the optimized version of the code, saving 0.01 s per image.

We used the same technique to study the original scripts and get some ideas of what parts should be optimized. We also used the `memory_profiler` package to understand the memory usage, without finding significant problems. This is the reason we will focus the optimization in reducing the amount of processing time.

[6] https://github.com/rkern/line_profiler

2.2 Basic Optimization

A very simple yet effective optimization first step is using list comprehensions. For instance, in `DownloadScript.py` we needed to append a character element (the directory name) to a list of characters (the file names). A simple approach using a for loop took 91 ns, while using list comprehensions took almost 120 ns. The main advantage is not the small increase in speed, but the increased code readability.

Code snippet part of Original/DownloadImages.py

```
files_c_2 = []
for element in files_c:
        files_c_2.append(folder + "\\" + element)
```

[Output] 22.9 μs $\pm$ 860 ns per loop

Code snippet part of Optimized/DownloadImagesFunction.py

```
folder = os.path.join(os.getcwd(), "Downloaded_images", "Man")
files_c = os.listdir(folder)
```

[Output] 19.8 μs $\pm$ 1.95 μs per loop

Another intuitive approach is to use a computationally expensive process only when required. Just after downloading a set of images using DownloadScript.py, we need to check that all these images are valid and in a correct format. It is not rare for some images not to be downloaded correctly, in which case the best course of action is to remove them. It is very common, however, for some images being in different formats than JPG and PNG, which will create problems when doing the face recognition and gender classification. Furthermore, it is impossible to trust the file extension of the images, because we have encountered many files ending in ".jpg" or ".png" that in reality are written to other formats. Our initial approach was to read each image using `PIL.Image` to test their format, and then convert them to JPG if possible and, if we obtained an error during the reading or saving the image, delete it assuming the file was corrupted. In a folder with just 100 images, this took almost 0.3 ms per picture in the folder. However, reading the entire file is not necessary if we just want to check the image's format. In fact, the function `imghdr.what` form the built-in **imghdr** package recognizes the image file format just based on their first few bytes. Therefore, it makes sense to first detect the format based on the first few bytes and then, only if it is different from JPG and PNG, read the entire file with `PIL.Image` and try to convert it. This reduces the processing time to just 0.1 ms per picture. Altogether this doesn't sound like much, it can make a difference when we load almost 500 pictures per search term.

Code snippet part of Original/DownloadImages.py

```
for item in files_c:
        try:
                im = Image.open(item)
                if not im.format in ["JPG", "JPEG", "PNG"]:
                        filename, extension = os.path.splitext(item)
                        im.save(folder + '\\' + filename + 'corrected' + '.jpg')
        except:
                os.remove(item)
```

[Output] 23.1 ms $\pm$ 264 μs per loop

Code snippet part of Optimized/DownloadImagesFunction.py

```
for item in files_c:
        try:
                if not imghdr.what(item) in ["jpg", "jpeg", "png"]:
                        im = Image.open(item)
                        filename, extension = os.path.splitext(item)
                        im.save(folder + '\\' + filename + 'corrected' + '.jpg')
        except:
                os.remove(item)
```

[Output] 10.7 ms ± 59.1 μs per loop

Lastly, we should be cautious about using regex in Python. In most cases it is a useful and easy solution, but sometimes a bad use may cause big problems in terms of performance. Although it was not causing an important slowdown in `DataAnalysis.ipynb`, it is easier to perform better using string methods. Specifically, we found the number of pictured based on the filename in just 575 ns, when regex took 1.34 μs.

Code snippet part of Original/DataAnalysis.ipynb

```
re.search(r'\d+', s).group()
```

[Output] 1.34 μs ± 278 ns per loop

Code snippet part of Optimized/DataAnalysis.ipynb

```
s.split()[0].replace(".", "")
```

[Output] 575 ns ± 166 ns per loop

2.3 NumPy and Pandas

Another common optimization procedure is using `NumPy` instead of the built-in lists. `NumPy` provides a multidimensional array object that: (i) uses fewer memory, (ii) works faster, and (iii) implements optimized functions like linear algebra. That is the reason it is the standard for scientific computing, but also for many image processing libraries. The module `cv2` natively uses `NumPy` arrays to store images, and both `PIL/Pillow` and `Matplotlib` work very well together with `NumPy` because their representations are also similar. Although in our code we are not using `NumPy` very often, take into account the image-processing packages utilized in `DownloadImages.py`, `CropFaces.py`, and `DetectGender.py` are using it. In the code snippets bellow we show the clear performance advantages of using `NumPy` instead of built-in lists.

The following code snippets are shown for illustration purposes, and are not part of the attached files

```
X1 = range(100)
Y1 = range(100)
X2 = np.arange(100)
Y2 = np.arange(100)
```

```
result = [X1[i] + Y1[i] for i in range(len(X1))]
```

[Output] 87.9 μs ± 4.94 ns per loop

```
result = X2+Y2
```

[Output] 737 ns ± 143 ns per loop

Pandas is arguably the most used library for data manipulation and analysis. It offers very useful data structures and can perform reasonably well – although clearly less than `NumPy`. However, there are some errors that are easy to fall into. A very common problem comes with loops. A standard for loop can take a long of unnecessary time; in our case, a for loop in the `Data_Analysis.ipynb` that iterated over one of the datasets (e.g. `Datasets/Nurse.csv`) containing the names of the files and the corresponding neural network outputs took 6.48 s for processing one dataset. Using a for loop with the iterrows method took significantly less, 151 ms. However, for an additional speed-up we had to use `NumPy` instead of Pandas. The very useful `numpy.vectorize` function allows to use a user-defined function over a `NumPy` array, bringing all the performance advantages of using `NumPy`. In this case, the script just took 559 μs.

The following code snippet is shown for illustration purposes, and is not part of the attached files

```
df['number'] = 0
for i in range(len(df.file)):
        df['number'][i] = re.search(r'\d+', df.file[i]).group()
```

[Output] 6.86 s ± 296 ms per loop

Code snipped part of Original/Data_Analysis.ipynb

```
df['number'] = 0
for i, row in df.iterrows():
        df.loc[i, 'number'] = re.search(r'\d+', row['file']).group()
```

[Output] 111 μs ± 8.13 μs per loop

Code snipped part of Optimized/Data_Analysis.ipynb. Modified version using lambda x: re.search(r'+', x) instead of lambda x: x.split()[0].replace(".", "") for comparison with the previous examples

```
vfunc = np.vectorize(lambda x: re.search(r'\d+', x))
df['number'] = vfunc(df.file.values)
```

[Output] 15 μs ± 2.2 μs ms per loop

2.4 Multiprocessing

Python was designed with simplicity in mind, and with this philosophy the built-in global interpreter lock (GIL) prevents the execution of multiple threads. However, running multiple operations at the same time is possible with the packages `threading` and `multiprocessing`. The main difference is that `threading` is more lightweight, and the threads run in the same memory space, while in `multiprocessing` the processes have separate memory and, very importantly, can run on different CPUs. Considering most computers nowadays include dual-core or quad-core CPUs, multiprocessing allows to bypass the GIL and use all this computational power. Therefore, is recommended for CPU intensive tasks. Nevertheless,

starting a threat is faster than starting a process and consumes less memory, so it is generally recommended for I/O bottlenecks.

The slower part of the code presented in the first section is, without doubt, downloading the Google Images results. Having to run this code for multiple search terms sequentially at approximately 11 minutes per search term is a clear I/O bottleneck. However, it is important to note that the `google-images-download` uses `Selenium` when downloading more than one hundred pictures per search term, and multithreading is not recommended with `Selenium` because WebDriver is not thread-safe. Therefore, we preferred to use the `multiprocessing` module. The implantation was more complex than expected, and required to create a function `DownloadImagesFunction` inside the Python file `DownloadImagesFunction.py`. This function was imported in the `DownloadImages.py` script and called inside the `pool.map` function. Note that this complex setup may only be required on Windows, and in other systems defining the `DownloadScript` inside `DownloadScript.py` may work. The results were impressive. Before, it took around 11 m to scrape the first 500 results for a certain search term. Now, in approximately 13 m we can download the same amount of results for 4 search terms in a quad-core CPU.

Detecting the faces and inferring gender are two very computationally expensive tasks that need to be performed. Therefore, both could potentially run much faster using all the cores available to the computer. We followed a similar approach to what was described in the previous paragraph. For cropping faces we created a function named `CropFacesFunction` inside a `CropFacesFunction.py` Python file, and imported and called it using `pool.map` in `CropFaces.py`. Before adding multiprocessing it took around 7 minutes to detect the faces for 500 photos. Now, we can process 500 images for 4 search terms (i.e. 2000 pictures in total in almost 9 minutes.

The same was done in `DetectGender.py`, which imported `DetectGenderFunction` from `DetectGenderFunction.py`. Importantly, we the code that loaded the Caffe classifier was put before (instead of inside) the `DetectGenderFunction` function so that code snippet was executed just once when importing `DetectGenderFunction.py` in `DetectGender.py`. If it was put inside `DetectGenderFunction` that would mean the classifier would reload each time the function is called (i.e. once for each search term), which will be an unnecessary waste of processing time. initially, it took approximately 6 minutes to infer the gender of the faces corresponding to 500 search results of a certain search term. With multiprocessing, this can be done for four search terms in 7 minutes.

2.5 GPU and CUDA

`Caffe` supports GPU-Acceleration using CUDA. This parallel computing platform allows to use the GPU for general-purpose data processing instead of the original purpose of this component, which is to improve the performance of video and graphics. More specifically, Caffe takes advantage of the boost provided by cuDNN, a library that makes use of CUDA and provides fast implantations for standard procedures related to the training and use of deep neural networks. Therefore, using CUDA with Caffe allows to take advantage of a powerful GPU, which for training neural networks can mean doing in a few hours something that in a CPU could take entire days. However, its advantages for inferencing tasks tend to be more modest. We are using, as we specified in the algorithm description, a pre-trained network to classify the gender of faces, and therefore our expectations were a moderate increase of performance, if any.

After setting up CUDA[7] and cuDNN[8], installing and setting up a Caffe version compatible with CUDA was not more complex than installing the CPU-only version in Windows (see the

[7] `https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html`
[8] `https://docs.nvidia.com/deeplearning/sdk/cudnn-install/index.html`

Windows Caffe branch on GitHub[9] for instructions). Once installed this GPU-compatible version of Caffe, we run `DetectGender.py`. However, we found a small decrease of performance: it took an estimate of 5 m 17 s, more than the 4 m 41 s that the CPU-version needed. There are different potential explanations for this. The first one is that Caffe was running on a Nvidia 940MX laptop GPU, one of the less powerful GPUs that support CUDA. Using a more powerful GPU will therefore increase the performance. Another possibility is a software problem related to the CUDA or Caffe installation (similar problems have been reported on Github[10]).

3 Conclusion

We started with code that already performed well, and therefore with limited optimization options. The image recognition tasks were already using C++-based libraries, and the data extraction had a I/O bottleneck that could be reduced. Even with this, multiprocessing almost fourfold reduced the time to scrape the images, detect the faces, and infer their gender. Additionally, some changes also made the code more efficient, particularly the data analysis part. However, we found that using the GPU didn't increase the performance of the Convolutional Neural Network in this case.

4 References

Caliskan, A., Bryson, J. J., & Narayanan, A. (2017). Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334), 183–186.

Candidate 1033256. (2019). Google Images search results reflect gender stereotypes. *Assignment for Applied Analytical Statistics* [unpublished].

Garg, N., Schiebinger, L., Jurafsky, D., & Zou, J. (2017). Word Embeddings Quantify 100 Years of Gender and Ethnic Stereotypes. *PNAS*, 115(16), E3635–E3644

Geitgey, A. (2018). Face Recognition. *GitHub*. Retrieved from https://github.com/ageitgey/face_recognition

Kay, M., Matuszek, C., y Munson, S. A. (2015). Unequal Representation and Gender Stereotypes in Image Search Results for Occupations. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*, 3819–3828.

King, D. E. (2009). Dlib-ml: A Machine Learning Toolkit. Journal of Machine Learning Research, 10, 1755–1758.

Levi, G., & Hassner, T. (2015). Age and gender classification using convolutional neural networks. 2015 *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 34–42.

U.S. Department of Labor, B. of L. S. (2018). Employed persons by detailed occupation, sex, race, and Hispanic or Latino ethnicity. Retrieved from https://www.bls.gov/cps/cpsaat11.htm

Vasa, H. (2018). Google images download. *GitHub*. Retrieved from https://github.com/hardikvasa/google-images-download

[9] `https://github.com/BVLC/caffe/tree/windows`
[10] `https://github.com/BVLC/caffe/issues/4791`