OpenMP and CUDA Parallelization of Viola-Jones Algorithm

1st Fabio Steven Tovar Ramos
National University of Colombia
Bogotá, Colombia
fstovarr@unal.edu.co

2nd Miguel Ángel Ortiz Marín
National University of Colombia
Cartagena, Colombia
miaortizma@unal.edu.co

***Abstract*—This paper describes an optimization of the Viola-Jones object detection framework with parallel computing frameworks OpenMP and Cuda. Viola-Jones framework uses a varied set of algorithms, namely Haar-like features along with an integral image representation for rapid calculation, a modification of the machine learning algorithm AdaBoost for selecting a small number from larger sets and a method for combining classifiers in a cascade which allows for negative samples to be quickly discarded. Parallelization efforts are focused on the feature application portion of the training phase. The OpenMP version of the algorithm is straightforward and achieves a 3x speedup in contrast to the sequential version. Furthermore, the CUDA version requires a different approach, so three methods are described, the last of them being very careful with memory management and using batch processing, obtaining an 18x speedup over the sequential version.**

***Index Terms*—face detection, machine learning, parallel computing**

I. Introduction

The necessity of doing tasks and calculations that require a significant amount of time is a problem presenting from the start of the computation era. There are several proposals from the hardware and software approaches to overcome this problem. Some of them have been broadly developed in actuality, allowing scientists, researchers, designers, among others, to be supported by these technological tools to do their work more accurately and efficiently.

This document contains some information about using parallel computing techniques to get a better performance in the training of a very well-known machine learning framework: The Viola-Jones object detection framework. We will take advantage of the computational resources to increase the speed of training of this algorithm.

We explored OpenMP and CUDA as possible techniques to distribute this job in several computational units. Later, the results obtained by doing these experiments will be analyzed and compared. Furthermore, we will propse future work regarding parallelization, specifically applied the Viola-Jones algorithm.

A. Viola-Jones Object Detection Framework

The Viola-Jones object detection framework is famously known for being used as the first real-time face-detector. The original paper [1] counts with around 22068 citations as of 2020. It introduced Haar-like features and a modification of AdaBoost in which a classifier is made out of a single feature. The algorithm continues to be used for face-detection (a binary classification problem). The popularity of the algorithm lies in the use of simple features that allow fast calculation. Though the framework is mainly used for face-detection, it may also be used to detect arbitrary objects. We proceed to define the elements of the framework.

1) Haar-like Features: The algorithm's first element is the Haar-like features, named due to its similarity with the Haar wavelet. Before defining what a Haar-like feature is, we must mention that from here on after every image is assumed to be in gray-scale. In case one has a set of RGB images, it can be transformed to gray-scale with the Gleam scale since we may assume that color information is of no use to the face-detection task [2]. A Haar-like feature takes into account a set of adjacent rectangles and assigns a sign to each of them (i.e., positive or negative). The feature is thus calculated as the sum of every point in each rectangle. Every point is either summed or subtracted from the total sum according to the sign of the respective rectangle. The intuition behind the Haar-like features is that it allows for detecting common regions that have a difference in brightness. For example, in general, the area of the eyes is brighter than the front of the nose area of a face. We define a total of 5 features, as is seen in figure 1.

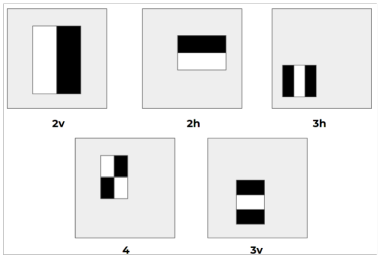

Fig. 1. 5 Haar-like features with names related to the orientation and number of rectangles. Black regions are substracted and white regions are added.

The smallest representation of each of the features may be defined in matrix form as follows:

$$\mathcal{F}_{2,h} = \begin{bmatrix} 1 & -1 \end{bmatrix} losspecs, numerodecoresyesoF_{2,v} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Identify applicable funding agency here. If none, delete this.

$$\mathcal{F}_{3,h} = \begin{bmatrix} -1 & 1 & -1 \end{bmatrix} \quad \mathcal{F}_{3,v} = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \quad \mathcal{F}_4 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

These matrices may be scaled (horizontally, vertically, or both) or positioned in all the legal positions (without any point outside of the image). Note that we must define the size to which the features are applied. This term is known as the window size. The final application of the framework consists of applying multiple of these features in cascade to all the possible sub-windows of an image. The window size determines the size of the faces that may be detected, though a trained classifier may be scaled to detect bigger or smaller objects.

For example, if the window size is 19, we end up with around 60 thousand possible features. This number is unfeasiable for real-time applications and thus should be reduced. First, we are going to explore a trick which allows for $\mathcal{O}(1)$ calculation of an arbitrary feature with $\mathcal{O}(N^2)$ precalculation of an integral image and only four memory accesses for a single rectangle. We will also see how we can use a modified version of AdaBoost to select a set of features to build a "strong" classifier and train a single "weak" classifier based on a single feature.

2) Integral Image: We define an integral image $\mathcal{I}^*$ of an image $\mathcal{I}$ as follows:

$$\mathcal{I}^*_{x,y} = \sum_{i=0}^{i=x} \sum_{j=0}^{j=y} I_{i,j} \tag{1}$$

That is, the (x, y) point of an integral image is the cumulative sum of all the points inside the rectangle $[(0, 0), (x, y)]$. It can also be seen as a discrete integral. With this, we may calculate the sum s of an arbitrary rectangle with the following relationship:

$$\begin{aligned} s(x_{left}, y_{top}, x_{right}, y_{bottom}) = \mathcal{I}^*_{x_{right}, y_{bottom}} \\ - \mathcal{I}^*_{x_{right}, y_{top}} \\ - \mathcal{I}^*_{x_{left}, y_{bottom}} \\ + \mathcal{I}^*_{x_{top}, y_{left}}. \end{aligned}$$

Assuming that $(0, 0)$ corresponds to the top-left corner, note that we are just removing the points to the left and above the desired rectangle, but we must add back the upper-left region as it has been subtracted twice. With the ability to calculate any rectangle in $\mathcal{O}(1)$ time we can calculate any feature $f(x)$ in $\mathcal{O}(1)$ as well.

3) Weak Classifier: We define a weak classifier $h(x)$ as follows:

$$h(x; f, p, \theta) = \begin{cases} 1 & \text{if } pf(x) < p\theta \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

A weak classifier is based on a feature f and is parametrized with a polarity p and a threshold θ. A weak classifier is a linear classifier; given a fixed threshold, it classifies all samples with a feature value under it as positive and all above as negative. The direction of the classifier may be inverted by setting the polarity to -1.

For a given threshold θ we have S^+ the number of positive samples with feature value under θ and S^- the number of negative samples with feature value under θ. We select the polarity and threshold, which minimizes the following expression:

$$\epsilon = \min \left(S^+ + \left(T^- - S^- \right) \quad , \quad S^- + \left(T^+ - S^+ \right) \right). \tag{3}$$

Note that the first term is associated with labeling all samples below the threshold as negative, while labeling all above as positive and vice versa. The training may be implemented quickly by trying each threshold in order, i.e., one calculates the feature value for every sample, orders the list of values with any $\mathcal{O}(n \log n)$ algorithm. Then the terms S^+ and S^- may be calculated easily as well as the error ϵ in $\mathcal{O}(1)$. This way, the sorting algorithm runtime bounds the training.

4) Strong Classifier: A strong classifier $C(x)$ is a linear combination of weak classifiers:

$$C(x) = \begin{cases} 1 & \text{if } \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The weights α_t are obtained from a standard AdaBoost procedure. AdaBoost is an implementation of boosting methods. Boosting methods train an ensemble of classifiers which complement each other. The basic idea is to give each training sample a weight towards the training error. The weight of correctly classified samples is reduced, and the weight of those incorrectly classified is increased. This way, classifiers are iteratively added to the ensemble focusing on incorrectly classified samples, as shown in figure 2.

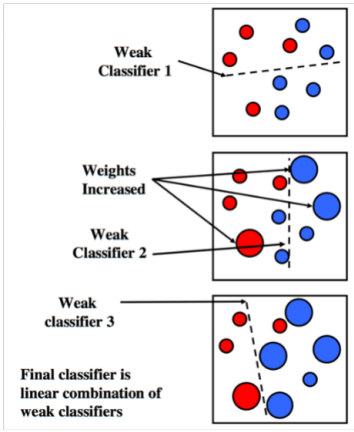

Fig. 2. Boosting procedure source: [3]

5) Cascade Classifier: Coupled with the ability to train a strong classifier, we can build a cascade classifier. A cascade classifier is one of the most critical elements of the Viola-Jones object detection framework. The objective is to apply

a sequence of models of increasingly complex features and stop if a sample fails any of the models criteria. A cascade classifier's main advantage is that it can discard many subwindows while using a tiny number of features, which are also rapidly calculated. Take into account that this framework does not exclude other types of models. Differents approaches such as deep learning could be attached as a final step of a cascade classifier. Classifiers with small number of features, small false positive rate and high recall are desirable in the front of the cascade.

A cascade classifier may be interpreted as a degenerate decision tree. This can be seen in figure 3.

Fig. 3. Cascade Classifier source: [3]

B. Distributed Computing Techniques

1) OpenMP: OpenMP is an API that allows writing computer applications using multithreading in an easier way than conventional methods. This library provides an interface to avoid communicating directly with the operative system, which is essential when to use applications over a broad range of devices [4].

OpenMP was designed to facilitate coding in a multithreading paradigm in languages such as C, C++, and FORTRAN. It is important to note that OpenMP works under a shared memory model and a fork-join model. The latter is a paradigm that seeks to divide the most expensive tasks by fork and then merge the results when joining. OpenMP aims to be global and, in a certain way, standardize parallel programming, as seen by the great variety of supported platforms. Some of the most important of these systems or compilers are ARM, GCC, Oracle, IBM, Intel.

2) CUDA: GPUs (Graphics Processing Units) are specialized units that allow rendering high-resolution 3D graphics in real-time, but they may have more applications since they are highly parallelizable multicore systems, which allows handling large blocks of data efficiently [5].

CUDA (Compute Unified Device Architecture) is a parallel programming API and platform developed by NVIDIA that enables the use of NVIDIA GPUs for general purpose applications. The API is designed for programming languages such as C, C++, Fortran, and Python. In a simplified way, an NVIDIA GPU has the following multiprocessor architecture, which will later facilitate our programs' design.

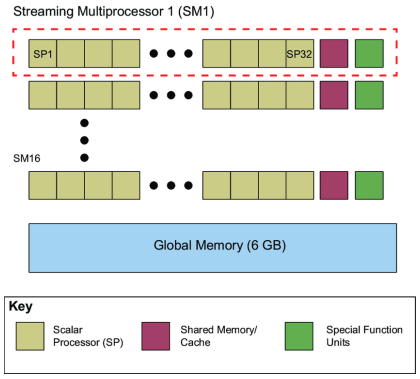

Fig. 4. Simplified diagram of a GPU architecture

As is shown in figure 4, there is a global memory for the device, the size of this global memory depends highly on the GPU we are working on, and the primary function of this part is to serve as a place to store temporal data to be used from the processing units to read and write, depending in the task that they have to do. These processing units are clustered in bigger units called multiprocessors, and by each multiprocessor, there is a shared memory that allows increasing the speed in the communication between threads as long as they are in the same multiprocessor.

II. Methodology

We implemented a version of Viola-Jones taking into account all the details already discussed in the present document. After doing that, we identified a couple of important sections that have the potential to optimize through parallelization. The first is calculating the features over the data set as a part of the training process. The second one is the application of the resulting model to a group of images.

However, they are very similar problems. They even could be solved by the same algorithm because applying the resulting model is to apply a single feature or a group of features to an image. Therefore, we focused on solving this problem on the training side, which is the most general solution. The code of our implementation is available in a Github repository.

A. OpenMP

As previously mentioned, parallelization with OpenMP is easy and straightforward. We apply one pragma in the loop used for applying all the features to one single sample.

In concrete, we add the directive :

```
#pragma omp parallel for
```

before the for loop.

The analysis of the current document requires the possibility to change the number of cores to be used in the parallelization of this section of the code in order to see the behavior of the time according to the computational resources that we could have. Fortunately, the `pragma` directive has the attribute `num_threads`, whose parameter determines the number of threads to be executed.

B. CUDA

We designed three different ways to solve this problem with this paradigm. In the following paragraphs, we described the general idea over the three solutions, and in the next section, it is possible to find a complete analysis of the results that we obtained with each of these solutions in terms of processing time.

1) First approach: The main idea was processing image per image in the GPU. By that, we had to allocate the memory for the image, copy the image to the device, allocate memory for the features to be calculated, copy them to the device, and allocate memory to save the results of each feature's application over the image. In 5 is possible to see a diagram of the previous behavior.

It should be noted that this diagram represents the behavior of the algorithm per each image.

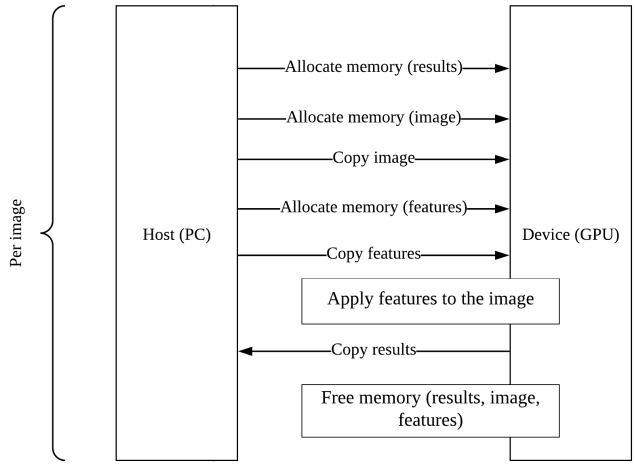

Fig. 5. Diagram of the first approach to solve the problem in CUDA

2) Second approach: Given that the features to calculate over the data set are invariant through time, it is unnecessary to copy them to the GPU for each image. We need to copy them once and access these resources when it is necessary. Furthermore, it is possible to reuse the memory for the results and the image instead of allocating and deallocating them for each image. A diagram for this simple but effective optimization could be seen in figure 6.

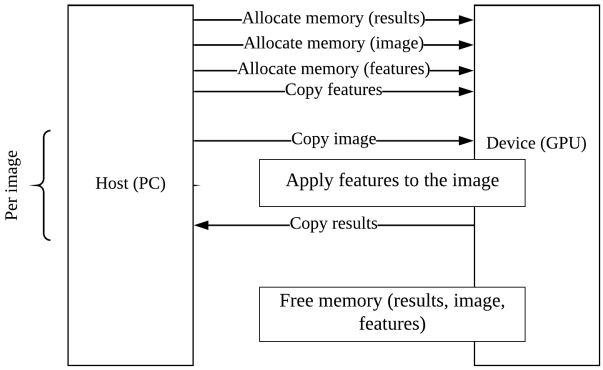

Fig. 6. Diagram of the second approach to solve the problem in CUDA

3) Third approach: To take advantage of the GPU's amount of memory and its capacity to do massive parallel operations in a short time, we decided to adapt the algorithm to process images by batches. Therefore, we query the amount of free memory of the GPU before starting the processing, and according to the amount of memory to be used in the results, images, features, and the occupation factor, the batch's size is calculated. The relation of these variables to determine the batch size is shown in (5).

$$batch = \frac{\alpha * free - features}{image + output} \tag{5}$$

Where α is the occupation factor and it should be between 0 and 1, $free$ is the amount of free memory in the CUDA device, and $features$, $image$ and $output$ are the sizes that these elements will use in memory. To estimate this values, we used the expressions showed in (6), (7) and (8). Here n_f is the number of features, d_f the dimensions of each feature, ii_w and ii_h are the width and height of the integral image. The scalar at the beginning of each expression (s_f, s_{ii}, s_o) shows the occupation in memory of each data type in the language.

$$features = s_f * n_f * d_f \tag{6}$$

$$image = s_{ii} * ii_w * ii_h \tag{7}$$

$$output = s_o * ii_w * ii_h * n_f \tag{8}$$

At the end, the general structure of this approach is shown in the following figure (7).

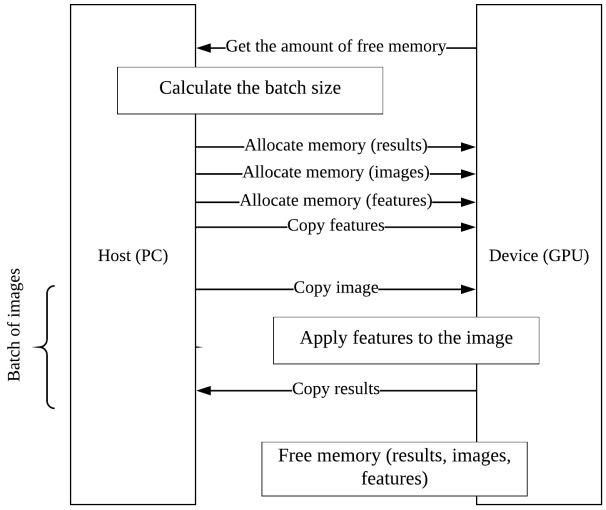

Fig. 7. Diagram of the third approach to solve the problem in CUDA

III. Results and Analysis

The time that the application took calculating the features for each image of the data set was measured for each method we proposed before. To get better results, we measure the duration of this function ten times for each specific configuration to be quite sure about the values we were getting and reduce the results' noise.

An excellent way to compare this kind of models is through the speedup, which is the relationship between an algorithm's sequential time and the time obtained by the parallel version of the same algorithm. That relationship is described in (9).

$$speedup = \frac{t_s}{t_p} \tag{9}$$

All the results are based on a machine with a intel core i7-8850 CPU @2.60GHz with 12 cores and a Quadro P2000 GPU with 4GB of memory. We are applying a total of 63960 features to a sample of 1000 19x19 images from the CBCL Face Dataset 1 used in [6].

A. Sequential

Our sequential C++ implementation takes 17.34 seconds on average to process the whole sample. The implementation is based on C++ standard libary vectors. During development one major improvement was obtained by using the keyword `const` and passing by reference with `&` where possible.

B. OpenMP

The results of the OpenMP approach are presented in figure 8. The results are very satisfying, given the straightforward application of OpenMP. The speedup curve plateaus near the expected value since the hardware used for running the tests has 12 CPU cores.

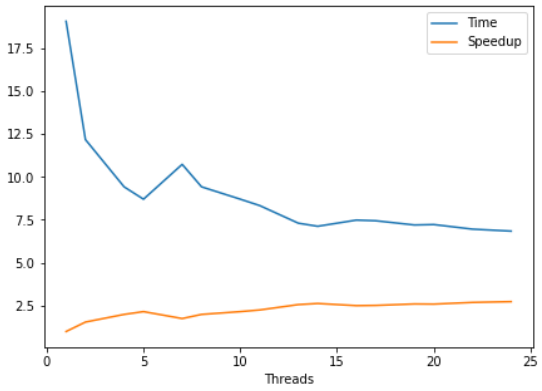

Fig. 8. Time and speedup with OpenMP

C. CUDA

The results of the first approach are presented in 9. It is possible to see the bad performance of this solution. The obtained time is higher than the times in the sequential version. Furthermore, while the number of threads was increasing, the time tended to increase, which shows a problem with the solution's formulation.

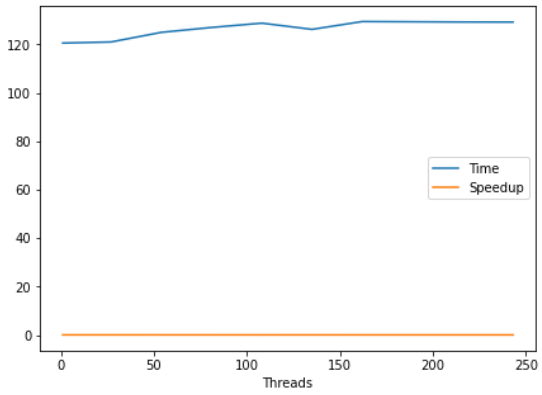

Fig. 9. Time and speedup in the first approach

The reason for getting these results is the wrong strategy used. Some operations that were necessary just once were repeated many times for each image. Also, `cudaMemCpy`, `cudaMalloc` and `cudaFree` are methods with a huge impact in terms of performance. The communication between PC and GPU is costly. That is the reason to avoid this process as much as possible.

On the other hand, the first optimization results, i.e., the second approach, were much better than the previous ones. In 10 the plot shows the improvement in processing time over the same data set. As the number of threads grows, the time decreases until it reaches a speedup of 2.2 approximately. That is a good sign, which proves the above paragraph's explanation about the importance of avoiding repeat calculations and being very careful with the communication with the device.

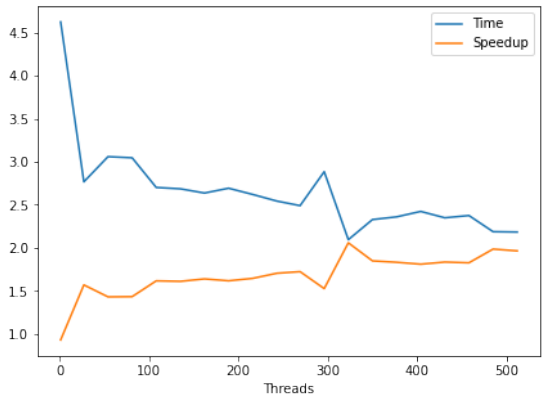

Fig. 10. Time and speedup in the second approach

The third approach far exceeded the results of the previous experiments. Working by batches and using a memory occupation factor of $\alpha = 0.8$, we got a speedup of 17.5 approximately over the same data set. The time was around 0.9 and 1 seconds.

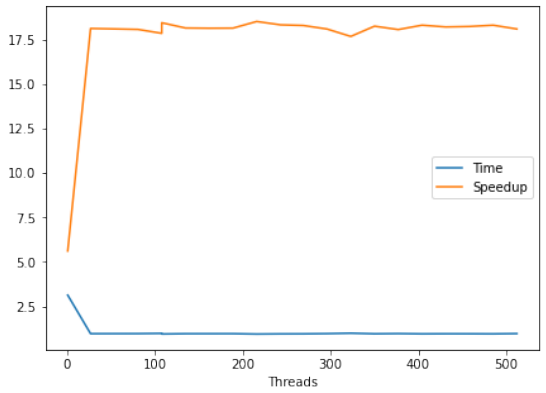

Fig. 11. Time and speedup in the third approach

Finally, the best results for the sequential algorithm and the parallel versions in OpenMP and CUDA are presented in the following table.

		CUDA		
Sequential	OpenMP	First approach	Second approach	Third approach
17.343880	5.500719	120.521115	2.038841	0.950823

Fig. 12. Best times in seconds obtained from the experiments

IV. Conclusions

When working in paralleling and distributed computing, it is essential to have strong knowledge about the background of the tools that we are pretending to use and have some insights about how is working the hardware associated with these tools since the design of the algorithm could highly depend on them.

In the case of OpenMP, it is necessary to know how the operative system handles the threads and the architecture of the CPU to understand some issues that could arise related to the hardware, such as false sharing.

On the other hand, CUDA is a powerful tool that allows optimizing certain kinds of algorithms. It is also necessary to understand a little about the communication between the CPU and GPU and read about acceptable practices and common problems of people who work in this area to prevent some issues.

Finally, the CUDA approach shows a good performance in solving this problem, by that it is possible to use this tool in the training phase of the Viola-Jones algorithm and the prediction stage. Even this algorithm could be used to identify objects from videos in real-time because of its good performance with an acceptable frame rate.

References

[1] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[2] Christopher Kanan and Garrison W Cottrell. Color-to-grayscale: does the method matter in image recognition? *PloS one*, 7(1):e29740, 2012.

[3] Anurag Jain. Computer vision – face detection, Jun 2019.

[4] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[6] M. Alvira and R. Rifkin. An empirical comparison of snow and svms for face detection. A.I. memo 2001-004, Center for Biological and Computational Learning, MIT, Cambridge, MA, 2001.