Undergraduate Implementation of the Viola-Jones Face Detector

Ryan Orf
University of Missouri – Columbia Department of Computer Science
CMP_SC 4650/CMP_SC 7650/ECE 4655/ECE 7655 Digital Image Processing – FS2021 Final Project Report
Columbia, MO, USA
rconzc@umsystem.edu

Date: Monday, January 13, 2021

***Abstract*—Face Detection is a common application in the modern world, ranging from use in entertainment in the form of social media filters to use in identifying and monitoring criminals when an act of crime has occurred. The Viola-Jones Face Detector is specifically noteworthy, as they developed a highly successful pipeline in 2001. Using this pipeline as a starting point, I implemented a program that will detect a face based on a machine-trained dataset, implemented from scratch using images I found independently and filters I created.**

I. Introduction: Motivation and Background

The goal of this project was to implement a program that would detect a facing-forward face in an image by utilizing a dataset trained through machine learning. Throughout this paper, I will explain the pipeline I developed following the Viola-Jones Face Detection approach, the general process of how I went about implementing the code and what each step does, the results I obtained from experimenting with my small dataset, and the conclusion I reached on how I could improve upon my program in the future. I chose to implement Face Detection because Face Detection interests me and is massively prevalent in modern society. As mentioned in the Abstract, Face Detection has a great range of usefulness. This usefulness and prevalence are what makes this problem so important to solve, especially in more serious uses such as identification of crime committers. The scope for this program is simple: if an image has a face facing forward, detect the face within the provided image. There are many existing solutions for this problem, one being the function “detectMultiScale()” from the OpenCV Python library. The program uses built-in Python libraries OpenCV, NumPy, and SciKit, and the algorithm I implemented was developed in large parts thanks to “A survey on face detection in the wild: past, present and future” by Zafeiriou, Zhang, and Zhang [1], and *Understanding face detection with the viola-jones object detection framework* by Lee [2].

II. Image Processing Modules and Pipeline

A. Contributions and Innovative Elements

In terms of personal contribution towards this program, I developed much in terms of the Segmentation, Processing, and Post Processing, which will be shown in the pipeline in Section III. In terms of Segmentation, I created the Haar feature kernels, as opposed to using built-in feature cascades. I use the same four fundamental features; however, I have filter sizes ranging from 2x2 to 32x32 to account for face position relative to the foreground. For convolution, I called the OpenCV function “filter2D()”. For Processing, I used the built-in OpenCV function “detectMultiScale()” to obtain the ground truth images. However, I implemented the machine learning myself, using images I found online for a dataset, and training and predicting the system myself using SciKit’s “svm.SVC()”, “fit()”, and “predict()”. In Post Processing, I determined which pixels are apart of a face, and subsequently which pixels needed to be set to borders.

B. Sample Input Images

Fig 1. An example input image

Fig 2. Another example input image

C. Expected Outputs

The outputs derived from convolving the filters, obtaining the ground truth data, and predicting faces in the desired image all return nd.arrays.The desired overall output for this program is a red box displayed around the faces detected in the image.

Due to shortcomings in terms of the machine learning that we will later discuss, this program does not tend to draw a perfect rectangle. However, faces are properly detected, and red boxes are drawn around the faces.

D. *New Concepts Learned*

Most of the coding concepts in terms of the algorithm had already been established from class. However, I did have to learn a good amount about machine learning, how it works and how to implement it yourself. I also had to learn concepts about Face Detection, like Haar features.

III. Approach: Algorithms and Module Implementation Details

Before discussing the details of the program, it would be wise to discuss Haar features and Integral Images for those who may not be familiar. Haar features are binary filters meant to represent facial features that are convolved with the image in order to help detect whether a face exists [1]. Rather than convolving the filters with the image, one could use the image's Integral Image instead. An Integral Image is an image calculated during preprocessing where each pixel value is actually the sum of the previous pixels [2]. I did not implement the Integral Image approach in my own algorithm, however in Section V, I will discuss how this could make this program more useful.

A. *Pipeline*

In my pipeline, which will be included below, I define four stages: Pre-Processing, Image Segmentation, Processing, and Post Processing. Under my terms, Pre-Processing consists of reading in the images and improving upon the quality. In this case, the images were reshaped to be smaller. Image Segmentation consists of convolving the Haar feature kernels with the images and detecting the features. Processing involves training the data using machine learning by comparing the features in the image to the ground truth data of whether a face exists at the corresponding pixel value. To put it simply, Post Processing is drawing a border around the detected faces and displaying the result to the user.

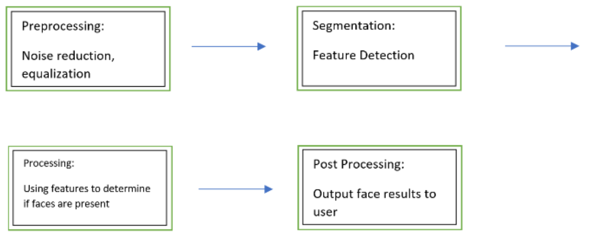

Preprocessing:
Noise reduction, equalization
Segmentation:
Feature Detection
Processing:
Using features to determine if faces are present
Post Processing:
Output face results to user

Fig 3. Project Pipeline

In comparison to the Viola-Jones Pipeline approach, my Pipeline consists of the same aspects, only categorized different. The Viola-Jones Pipeline defines its stages as Calculating Haar like Features, Calculating Integral Image (not applicable for this project), Developing the AdaBoost Algorithm, Creating The Cascade Classifier, Training, and Testing the Application [1]. The main differences are that I included reshaping in the Pre-Processing, I did not include the Integral Image, and my Pipeline stages are more grouped together.

B. *Program Process/ "Pseudocode"*

The first part of the program simply declares and initializes Haar feature kernels using NumPy to create the arrays. Next, each image that will be a part of the dataset is read into the program as a Grayscale Image. The image is scaled down to 11% the original size, and arrays corresponding to the training data and the training target are appropriately sized. The training data is a two-dimensional array, with the number of rows representing the number of features, and the number of columns representing the rows of each image times the columns of each image. Similarly, the training target is a one-dimensional array that is equal to the size of the number of rows in each image times the number of columns in each image.

Continued on, the code loop through each dataset image. Each image is convolved with each Haar feature using OpenCV's "filter2D()". The other bit of information required is the ground truth data. In order to obtain the ground truth data, OpenCV's "CascadeClassifier()" and "detectMultiScale()" are used on each image, and another binary array is created equal to the size of each image, with 0 representing no face detected at that pixel and 1 representing a face detected at that pixel.

Within each image, all of the pixels have to be looped through. Values for training data and training target are now assigned using the data from the features and the ground truth information. Once every image has been passed over in order to collect information for training, the program can use machine learning to train.

Training consists of using SciKit's "svm.SVC()" and "fit()" functions. Built-in function "svm.SVC()" creates the machine learner, and "fit()" trains the learner using the training data and training target previously calculated. Now that the dataset has been trained, machine learning can be used to predict a face in a prompted user-provided image.

The user-provided image must undergo the same precursors. A Grayscale Image must be opened and scaled down, and the image has to be convolved with the features. However, on this pass of the pixels in the image, instead of setting values for our training data and training target, we set values corresponding to an array about predicting images, using the feature data recently calculated.

Finally, the prediction can happen using SciKit's "predict()" function. The "predict()" function will return a binary array similar to the training target. The Grayscale Image must then be looped through. If the current pixel is a face pixel, that pixel in a colored copy of the input image is set equal to red. In order to calculate this, the scaling factor used earlier must be reversed engineered. That way, all of the pixels corresponding to a singular pixel in the scaled-down image are properly assessed.

One more loop over the pixels in the image check whether a pixel in the colored image is a border pixel to a face pixel. If this is the case, i.e. a pixel next to the current pixel in the colored copy is red, the current pixel is set to red, forming the rectangular border. Both the intermediate-colored copy and the bordered color image are displayed to the user for ten seconds, before the program terminates.

C. *Tools Used*

Of course, performing this algorithm and training this data could not be done without the help of built-in functions and images. For a dataset, I implemented my own using nine images of random faces I found online. For Packages I used in Python, I used OpenCV (cv2), NumPy (numpy), and SciKit (sklearn).

IV. Experimental Results and Discussion

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper; use the scroll down window on the left of the MS Word Formatting toolbar.

A. *Input and Output Images*

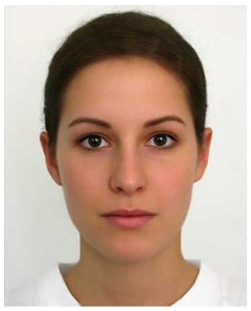


Fig 4. An example input image used earlier

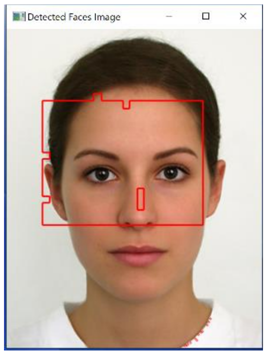


Fig 5. Example output from the input image in Fig 5.

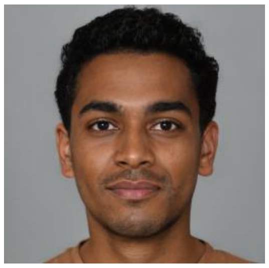


Fig 6. Another example input image used earlier.

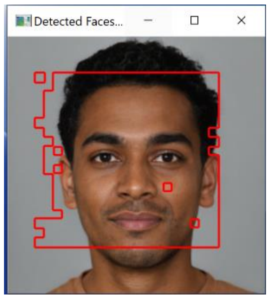


Fig 7. Another example output from the input image in Fig 6.

B. *Intermediate Images*

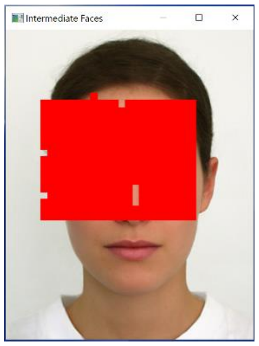


Fig 8. Example intermediate output for the input image in Fig 5.

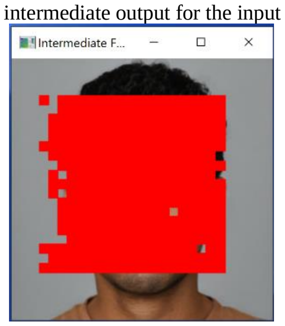
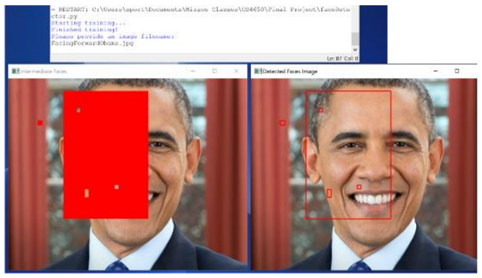


Fig 9. Another example intermediate output for the input image in Fig 6.

C. *Process from a User's Perspective and Performance*

The usability of the program from the user's perspective is incredibly straightforward. First, ensure your desired image is located in the same directory as the program. Then, run the program. The program outputs text to let the user know it is training. When the program is done training, the user is notified and prompted to input an image's filename. If an image cannot be found or loaded in, the user is notified and continuously prompted until the user quits or provides a valid image file path. As expected, after a short amount of time, the intermediate and final output images are displayed to the user for ten seconds before the program ends.



```
= RESTART: C:\Users\sport\Documents\Mizzou Classes\CS4650\Final Project\faceDete
ctor.py
Starting training...
Finished training!
Please provide an image filename:
FacingForwardObama.jpg
```

Fig 10. Example execution of program from the user's perspective.

As for the performance of this program, it is not optimized. Using the Integral Image would have likely made the program train and execute exponentially faster. With this implementation, I have determined that the complexity of the program is O(n^3), or O(n*n^2), where n is the number of features and n^2 is the number of samples, i.e., the number of pixels in all the images combined. Please find below an example of an optimized and more efficient Face Detection implementation using OpenCV's "detectMultiScale()" function.

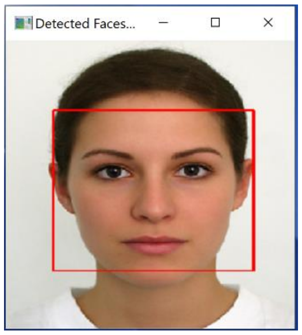


Fig 11. Built-in face detection output using the input image from Fig 5.

V. Conclusion and Future Work

Expectations were very different from reality when it came to implementing the final version of this project. I thought I would be able to implement a highly accurate (~90-95% accurate) face detector that would work on any image, similar to OpenCV's "detectMultiScale()" function. However, having implemented nearly every part of the pipeline myself, including the dataset and the machine learning, this task would simply not be accomplished. The program only correctly detects faces in the images used for training, and the face border is not as accurate as I would like, clipping off parts of the face and adding holes. However, since I implemented so much of the pipeline myself and only used built-in functions where I needed to, I am happy with this result.

The biggest problem I encountered came with machine learning, specifically SciKit's "sklearn.svm.SVC()" for training and predicting. A large complexity led to a slow runtime, which is what led me to scaling down the images in Pre-Processing. My original training implementation ran for over six hours with no prevail due to the sheer size of the images and the O(n^3) complexity. However, resizing the images also led to decreased accuracy in Processing and Post Processing, which is what led to holes in the facial border and parts of the face being clipped from the detector. Of course, the machine learner also does not perform well with images not related to the training dataset, since I had to implement such a small dataset from scratch.

My biggest takeaway from this project has to do with machine learning. I am lucky to say I understand how the Viola-Jones algorithm works; the logic makes sense. However, my end result was much less accurate than I expected due to the requirements of the machine learner and the size of the dataset I chose.

In the future, I would look to improve on a number of things in this program. For one, I would consider using a different machine learning implementation such as PyTorch. I would almost definitely use predefined datasets and pretrained machine learners, as this would help preserve accuracy, computation time, and hardware space. Finally, I would consider implementing the Integral Image approach, so as to save unfathomably on computation time. My program could be optimized in several regards, but ultimately it works on my dataset, and I have taken away many lessons learned in machine learning and digital image processing.

References

[1] Lee, S. (2021, August 11). *Understanding face detection with the viola-jones object detection framework*. Medium. Retrieved December 13, 2021, from https://towardsdatascience.com/understanding-face-detection-with-the-viola-jones-object-detection-framework-c55cc2a9da14.

[2] Zafeiriou, Stefanos, Cha Zhang, and Zhengyou Zhang. "A survey on face detection in the wild: past, present and future." Computer Vision and Image Understanding 138 (2015): 1-24, --- 449 citations.