

Viola-Jones Facial Detection Implementation

By: Phan Trinh Ha, Tim Jin and Kris Zhao, Kyle Pham

Motivation:

Facial detection is an incredibly important concept. Given the rapidly changing world towards AI monitoring, it is useful to look into the technology and algorithms driving the change.

There are a broad range of applications towards facial detection. The simple case is for social media. Tagging faces and friends in photos can create an interconnected network of friends and family quickly and easily. On the security side, as is the direction with a lot of security in the cities (I believe Seattle just approved of facial recognition use in security), facial detection and recognition can help identify threats.

While applications can be widely contested, facial detection can be an incredibly useful tool for home automation, assistants, phone security, etc...

Project Description:

Given the emerging technological applications to facial detection and the development of various algorithms, our project will be focused around implementing the popular Viola-Jones algorithm.

In order to keep the scope bounded for the class project, we wanted to recreate a well known algorithm ourselves utilizing python. Through the process, this will introduce us to the steps involved with facial recognition.

On top of this, we want to work with performance metrics for our algorithm. This will include running test images, checking error rates, and then comparing with other implementations.

Project Milestones:

Throughout the process, we hit certain milestones which helped us stay organized and keep everyone on track. These will simply be the anticipated steps in implementing our algorithm. Each component could then be completed and then brought back together.

A couple resources that we found to guide the breakdown of the parts can be found below.

The rough outline we are following is described in the following paper:

- <https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf>

Facial Classifier tutorial (with library functions we cannot use):

- <https://towardsdatascience.com/a-guide-to-face-detection-in-python-3eab0f6b9fcl>

Outline of Viola Jones Detection:

- https://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework

Integral Image (Kris Zhao):

In this phase, we will take our image that we will try and detect. We will then compute a summed area table representation of the image. In short, this allows for rapid application to the image. The reason for this is that we can calculate the sum of pixels using only constant time operations (adding / subtracting 4 numbers).

As such, utilizing the summed table will allow the facial detector to act very rapidly and in real time.

Integral image calculation:

- https://en.wikipedia.org/wiki/Summed-area_table

Haar Features (Timothy Jin):

Next, we will need to implement Haar filters in order to detect the facial structure. Since most faces have a set of defined features (like eyes, nose, mouth, etc), they will align fairly well with the basic functions of the Haar feature detection filter.

The issue with Haar features is that there are a variety of different features that can be detected within a window.

Haar Feature detection (Utilizing open CV):

- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html

Overview of Haar Features:

- <https://medium.com/analytics-vidhya/what-is-haar-features-used-in-face-detection-a7e531c8332b>

Adaboost Classifier (Kyle Pham):

In order to account for the variety of different detectable features, we will also need to implement an adaboost classifier. The adaboost classifier works by taking a set of weak classifiers such as threshold functions, then weighting a linear combination of the weak classifiers.

- Given example images $(x_1, y_1), \dots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.

- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.

- For $t = 1, \dots, T$:

1. Normalize the weights,

$$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^n w_{t,j}}$$

so that w_t is a probability distribution.

2. For each feature, j , train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t , $\epsilon_j = \sum_i w_i |h_j(x_i) - y_i|$.

3. Choose the classifier, h_t , with the lowest error ϵ_t .

4. Update the weights:

$$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$

where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.

- The final strong classifier is:

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

where $\alpha_t = \log \frac{1}{\beta_t}$

Table 1: The AdaBoost algorithm for classifier learning. Each round of boosting selects one feature from the 180,000 potential features.

(Pseudocode for the AdaBoost algorithm described in Viola Jones)

Through the learning algorithm, we will be able to slowly determine the best features to emphasize (Haar features) as well as modify weights on the combination of classifiers that utilize the features.

Adaboost Example in Python:

- <https://towardsdatascience.com/machine-learning-part-17-boosting-algorithms-adaboost-in-python-d00faac6c464>
- <https://www.datacamp.com/community/tutorials/adaboost-classifier-python>

Boosting overview:

- <https://www.python-course.eu/Boosting.php>

Cascading: (Phan Trinh Ha)

Finally, in order to find the true faces and not possible false features, we will need to run the results in a cascading fashion. What this means is we take the results for one level, then pass it into a stronger classifier. We repeat this and will classify each feature as a face if the features pass through each layer of the filter. Viola and Jones described this framework as collecting all of the previous false negatives from the previous partial cascade to the next cascade, and used the set of false negatives to train the next classifier in the cascade. The intuition behind this is that we are training on simpler images with less feature counts (simpler features/less weak classifiers) so that the time it takes for simple features to be detected reduces. However, with more complex background features, we can then pass on to further cascades, with more complex features (more feature counts/weak classifiers).

What this will accomplish is to only run the stronger classifier on a set of possible faces (sort of asking targeted questions at potential faces instead of asking targeted questions at every single possible image that could or could not be a face)

Visualization from the Viola Jones paper:

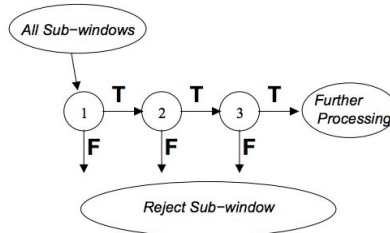


Figure 4: Schematic depiction of a the detection cascade. A series of classifiers are applied to every sub-window. The initial classifier eliminates a large number of negative examples with very little processing. Subsequent layers eliminate additional negatives but require additional computation. After several stages of processing the number of sub-windows have been reduced radically. Further processing can take any form such as additional stages of the cascade (as in our detection system) or an alternative detection system.

Cascading classifier for OpenCV:

- https://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html

Cascading Classifiers

- https://en.wikipedia.org/wiki/Cascading_classifiers

The Algorithm:

Aside from being broken into the previous sections, one of the important notes about the viola jones algorithm is the focus on background information.

Since faces have a very defined and standard feature set (eyes, nose, mouth...), they can be classified easier. We can look for features that match this to sample images and get a pretty good idea of what weights work with faces.

The issue is when it comes to the background. Instead of just eyes, or noses, there can be various other objects, such as posters, occlusion, trees, oceans, shops.... Because of this, one of the key components to the algorithm is to work on reducing the false positives in backgrounds. This can be accomplished by cascading the no-face dataset to get a very accurate set of features and weights for non-face images.

Implementation:

In terms of implementation, we have completed all of the project through python in standalone files. While jupyter notebooks are incredibly useful for the tutorial sense as well

as sequential explanations, utilizing standalone files provided a more streamlined process while developing more interwoven code.

In lieu of the jupyter notebook style, we have gone ahead and completed in-depth tutorial-like explanations along every step we went through inside our python files. **Please read through the python files as if they were jupyter notebooks.** These should provide adequate explanation as to our process and aid those looking to learn off of our code and how facial detection works.

Integral Image:

The integral image implementation has been completed within the `integral_image.py` file. The code should be self explanatory and comprehensive. The following functions have been implemented and remain crucial in the viola jones algorithm.

- `calc_int()`
- `region_sum()`

The main method utilized is just for testing purposes. The `integral_image.py` file does not need to be run by itself to work with our face detector.

Haar Features:

The haar features implementation is similar to integral images in that it is provided in a python file to be used in other locations. These functions can be imported and are utilized in the viola jones algorithm:

- `Class HaarFeature`
- `score()`
- `compute_region()`
- `ii_value()`
- `haar_features()`

Main methods have been implemented for the sole purpose of testing.

Modeling the classification:

Adaboost Classifier:

The adaboost classifier has been created within the `model.py` python file. This includes the various classes required of an adaboost classifier. We have defined the following classes which pertain to the adaboost classifier:

- `Class WeakClassifier`

- `classify()`
 - `train_id()`
- Class `AdaBoostModel`
 - `train()`
 - `classify()`

Cascading features:

Both of the classifiers in the previous section were also utilized when we implemented cascading of the feature sets. Within our cascading classifier class we implemented much of the same functions

- Class `CascadeClassifier`
 - `train()`
 - `classify()`

Util.py

Aside from everything defined within the `model.py` file, we utilized definitions for superclasses and functions that would be repeatedly utilized inside the `Util.py` file. This provided functionality such as importing / exporting data for each of the models, as well as quickly pulling in image data from folders.

Dedicated functions:

Aside from the mentioned classes, we also utilized the following methods within our `model.py` file. This includes functionality for processing data (explained in comments), picking classifiers, and training the data:

- `preprocess_data()`
- `_best_weak_classifier()`
- `_weak_classifiers()`
- `_training_data()`

Running:

The important thing to note is that we WILL be running the `model.py` data file. This file is where we will be generating and training our classifier. There will not be command line arguments passed in, but instead, each setting should be configured in the file before running *python model.py*.

1. Define the layers of the classifier (each integer is another layer, and the integer itself represents the iterations through the data).

2. Define the number of positive images pulled, and the number of negative images. There are approximately 2400 positive images, and 5000 negative images in the provided folders.
3. Specify the model filename for where the trained classifier will be saved (to be loaded at a later time)

After running the program, we will then get our .pkl file which stores our model classifiery. These can then be utilized for classification of faces (such as in our testing.py and webcam_classifier.py files).

Testing the models:

After we had completed the model and training, we went ahead and looked into testing our model. This was accomplished through the testing.py file. Inside this, we run through all of our training images and classify them based on the classifier we trained previously.

This allows us to calculate the false / true positive / negative rates and determine our ROC curve. The included functions are as follows:

- scores()
- model_confidence()

The *testing.py* file is intended for running. We do not have command line arguments but instead must modify the file itself. The modifications and configuration is as follows at the bottom of the document:

1. Define the model that you would like to run the tests on. This will be the .pkl file without the extension.

Afterwards, it will run through and generate all the statistics on the operations of the classifier. It will also pop up with an ROC curve to look at the behavior of the classifier passed to it.

Running and using the model:

Once we have statistical information about the model and we know that it is able to classify results, we can go ahead and utilize our model with current webcam feeds. This has been implemented within our *webcam_classifier.py* file. Here, the main parameters will be:

1. The model file (the file generated by the model.py file) specified at the beginning

2. The camera webcam number. This should default to 0 as the main camera but if multiple cameras are connected, they can be selected with other integer values.

After running, it should pop up a live feed of the webcam and squares over detected features (faces). This will continue in a loop and in order to close the program, we can go ahead and press the **escape** key.

In order to save the current image, we can press **s**.

Pitfalls:

While working through the implementation, we had encountered a couple issues that posed problems for our project.

Runtime:

One of the biggest issues is the runtime for the classifier. While we want to detect faces in images in general, the quicker functions run the easier it is to develop, test, and apply.

Training the classifier:

One of the most time consuming tasks ended up being the classifier. Since we are using a cascade classifier, running across thousands of images took up a lot of time. This meant a lot of testing and development required training on a very very small batch of images.

For example, once we trained on all the data, the process ran for up to 3 hours. If we were testing with classifications that took 3 hours, it would provide a major hurdle in developing our classifiers and models.

Fixes / Remedies:

One of the fixes we could apply to account for long runtimes is the utilization of nump parallelization. This was especially helpful in computing the Haar functions and scores. This provided a decent speed up, but when processing large datasets, it will always have a time component to it.

Real-time Tracking:

Another time slowdown that we encountered was when we were applying our model through the webcam. Since we wanted to provide real-time face tracking, iterating

throughout the entire image was incredibly difficult and time consuming. Especially to keep up with reasonable frame rates (such as 30 frames per second), we would have to essentially process the number of images per second.

While we are able to help, the camera processes images at approximately .5 seconds per image.

Fixes / Remedies:

The major fix applied here was to first parallelize the haar feature and scoring, but then modify the images that we are working with.

The larger the image, the more possible locations that our features may occur. As a result, we lowered the resolution of the image before classifying. This allowed us to increase the speed from 1-2 seconds each image, to just under .5 seconds for each image. While this isn't the optimal performance for facial detection, it does accomplish the primary goal when looking at live video.

Feature Sizes:

A final issue that was difficult to handle was the feature sizes. Because we were setting a haar feature size, some of the features detected might pertain to an aspect of a face but the image of the face itself may be too large or too small. As such, the entire feature will not be correctly "observed".

Fixes / Remedies:

Some of the possible fixes that we explored were to scale the image (lower the resolution so it's like we're applying a larger feature). We applied this within the webcam classifier and scan through unzoomed features, and then zoomed features of 2x the size. This gave reasonable results but given the pixelated nature of the image, can provide inconsistencies with the classifications.

This is still questionable and we are still considering a better way to approach this issue.

Sample Results:

For a sample runthrough, we run through working with the three files in the following order:

1. model.py
2. testing.py
3. webcam_classifier.py

model.py

Here, we can run through training a model. Most importantly, we can set the following parameters:

```
if __name__ == '__main__':  
    # Define the layers for the classifiers  
    layers = [2, 10, 20, 50]  
  
    # Define number of images for each set:  
    num_pos_images = 50  
    num_neg_images = 100  
  
    # Define the name for the saved model file:  
    model_name = 'cascade_50-100'
```

This will create a 4 layer cascade, with 50 images with faces, and 100 images without faces. This should be a function which can run in a shorter amount of time to illustrate the model.py functionality.

There have been other run throughs completed by us. For instance, the cascade_full took 3 hours to train on all the provided image data. Again, this does not correspond to being better or worse.

testing.py

Once we have our model data, we can then run it through the tester. Inside the file, we can modify the model we would like to load. Let's go ahead and run the test on the model we just generated:

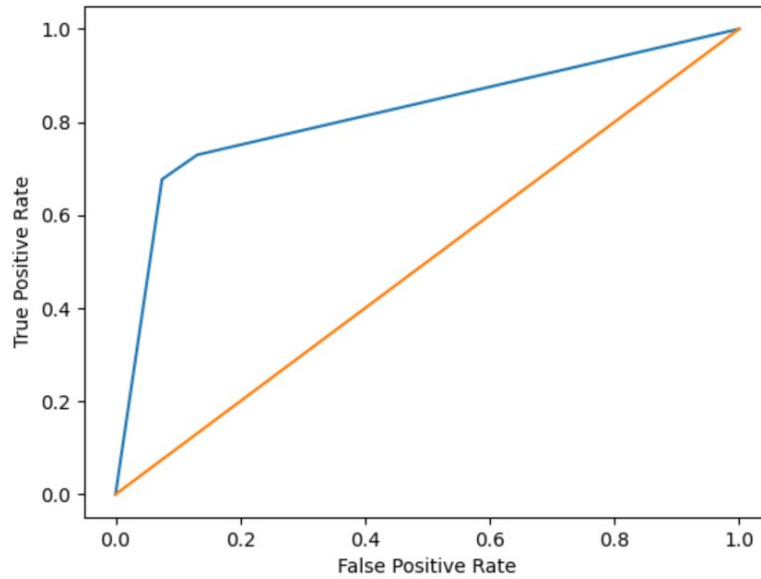
```
if __name__ == "__main__":  
    # Utilize the load method for any classifier to run test data  
    model = PickleMixin.load('cascade_50-100')
```

This can give us some of the information about how well our model is performing.

```

Classifying all images:...
100% |#####|
Calculating confidence results:
100% |#####|
False Positive Rate: 339/4547 (0.074555)
False Negative Rate: 785/2429 (0.323178)
Accuracy: 5852/6976 (0.838876)
Average Classification Time: 0.003665

```

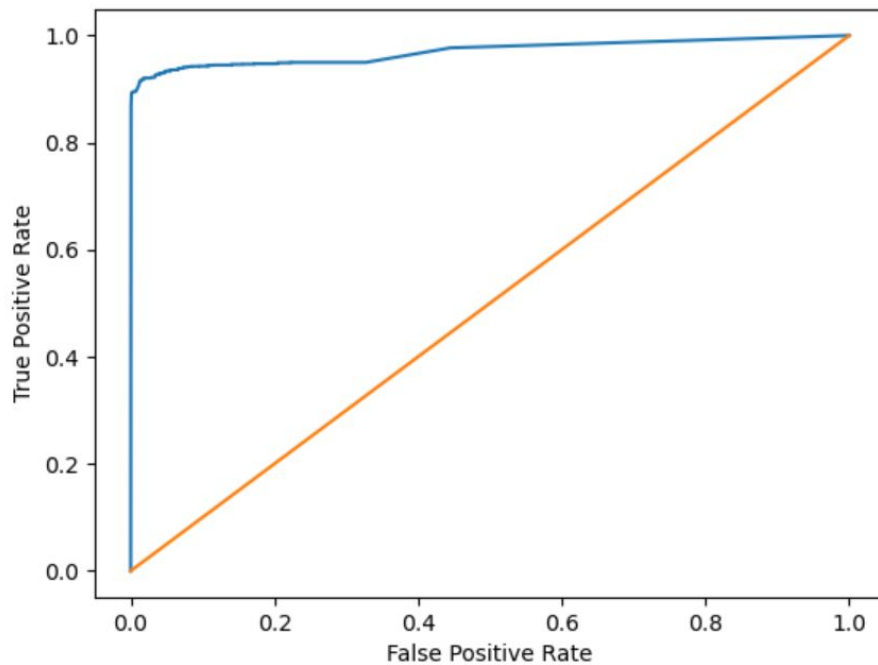


For comparison, we can also look at the cascade_full model.

```

Classifying all images:...
100% |#####|
Calculating confidence results:
100% |#####|
False Positive Rate: 1449/4547 (0.318672)
False Negative Rate: 122/2429 (0.050226)
Accuracy: 5405/6976 (0.774799)
Average Classification Time: 0.007989

```



Instead of having a lower false negative rate, the full dataset is much better at detecting true faces, but starts to suffer when it comes to the amount of false positives. As such, the total accuracy does drop when compared to the other model.

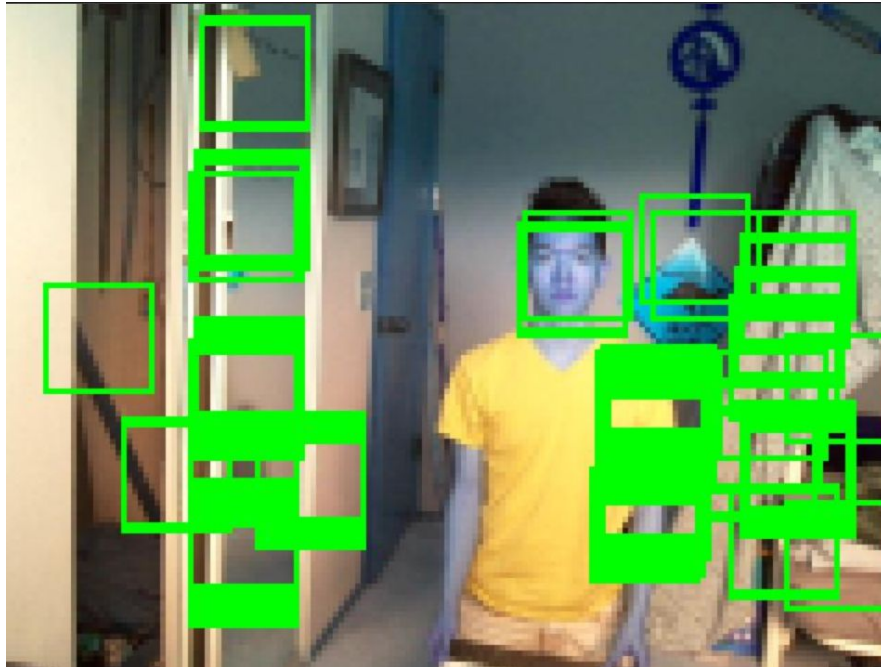
This can really go to show the tradeoffs between the positives and negative rates. We can now take both of these models and see how they perform in the webcam.

webcam_classifier.py

Let's begin with utilizing the first generated model file and the default camera.





```
if __name__ == '__main__':  
    # Specify the model file to be used and camera number  
    model_file = 'cascade_50-100'  
    camera_number = 0
```

Running this we can see the result:



Now this is far from perfect. As we can see there are still a lot of false positives (but it does detect the face correctly). While there are weird coloring issues, the images are converted to grayscale and should not play a role.

Analyzing the image, we can see that a lot of the false positives line up with some vertical features or changes across in light to dark. This would correspond to the Haar features used in facial detection (typical facial features). These features have been shown below from [Medium](#):

																																							
<table><tr><td>-1</td><td>-1</td><td>5</td></tr><tr><td>-1</td><td>-1</td><td>5</td></tr><tr><td>-1</td><td>-1</td><td>5</td></tr></table>	-1	-1	5	-1	-1	5	-1	-1	5	<table><tr><td>5</td><td>5</td><td>5</td></tr><tr><td>-1</td><td>-1</td><td>-1</td></tr><tr><td>-1</td><td>-1</td><td>-1</td></tr></table>	5	5	5	-1	-1	-1	-1	-1	-1	<table><tr><td>-1</td><td>5</td><td>-1</td></tr><tr><td>-1</td><td>5</td><td>-1</td></tr><tr><td>-1</td><td>5</td><td>-1</td></tr></table>	-1	5	-1	-1	5	-1	-1	5	-1	<table><tr><td>5</td><td>-1</td><td>-1</td></tr><tr><td>-1</td><td>5</td><td>-1</td></tr><tr><td>-1</td><td>-1</td><td>5</td></tr></table>	5	-1	-1	-1	5	-1	-1	-1	5
-1	-1	5																																					
-1	-1	5																																					
-1	-1	5																																					
5	5	5																																					
-1	-1	-1																																					
-1	-1	-1																																					
-1	5	-1																																					
-1	5	-1																																					
-1	5	-1																																					
5	-1	-1																																					
-1	5	-1																																					
-1	-1	5																																					
(1)	(2)	(3)	(4)																																				

In other backgrounds, results have also looked even more promising:

