Machine Learning
Viola Jones Face Detection Algorithm Implementation

Adam Holmes

May 2 2017

1 Design Decisions

1.1 Structure

The general structure of my implementation attempts to clearly expose which components of the description of the Viola-Jones (VJ) algorithm are being implemented and where, as originally described in [VJ04]. Here I describe the major functions of the implementation:

- Preprocessing
- Integral Image Formation
- Feature Generation
- Function: Train
- Function: adaBoost
- Function: bestLearner
- Function: setBigTheta
- Function: classifierCascade

Omitted are functions like calcFalseRates, which is responsible for generating false positive and negative statistics, and other similar less critical functions.

1.1.1 Preprocessing

Program main begins by loading in training and test data by reading and converting to grayscale using built-in numpy features. Next, sub-windows are generated on the test image by slicing the original image into 64×64 pixel blocks, iterating in strides of 8 pixels at a time. A supplementary list *coords* is created that holds a corresponding set of points (upper left and lower right) of each window, for use in drawing detected faces on top of the test image after processing.

1.1.2 Integral Images

Integral images are calculated by applying built-in numpy functionality *cumulative sum* on each of the training images, and each window of the test image. This function calculates the summed-area matrix of each image, and loads the training images into the global list *iimages*. At this stage, images are ready for processing.

1.1.3 Features

Haar-like features are created in two distinct nested for-loops. The two features chosen are two of the two-rectangle features, described in Figure 1. These features are constructed by selecting upper left and bottom right coordinates from the 64 $\times$ 64 pixel grid for each of the light and dark rectangles. A design decision made was to ensure that the area of both rectangles is equal, and that they are of the same dimensions. The dimensions of rectangles across different features is allowed to vary, but within a single feature the light and dark rectangles are enforced to be the same size. Features were selected at a stride of STRIDE for a total of TOTALFEATURES. Once features are generated, they are loaded into the global list *featuretbl*.

1. Type 1:
   Corresponds to B on Figure 1.

2. Type 2:
   Corresponds to A on Figure 1.

1.1.4 Function: Train

This is the outermost function called by the program main, and is responsible for training and constructing each classifier and the stages of the classifier cascade. The train function operates as follows: This function does not assert any threshold values other than the fact

Algorithm 1 Train Implementation

```
• Prepare lists of examples, and stopping values corresponding to 1% of the original
  negative examples
• While the number of examples $\geq 0.51\times$ number of original examples:
    – Run adaBoost and construct a hypothesis along with a value of $\Theta$
    – Eliminate correctly predicted backgrounds from example set
    – Append hypothesis and $\Theta$ to the classifier cascade set
```

that termination of the function occurs when the total number of examples is reduced such that there are only 1% of the negative examples remaining in the set.

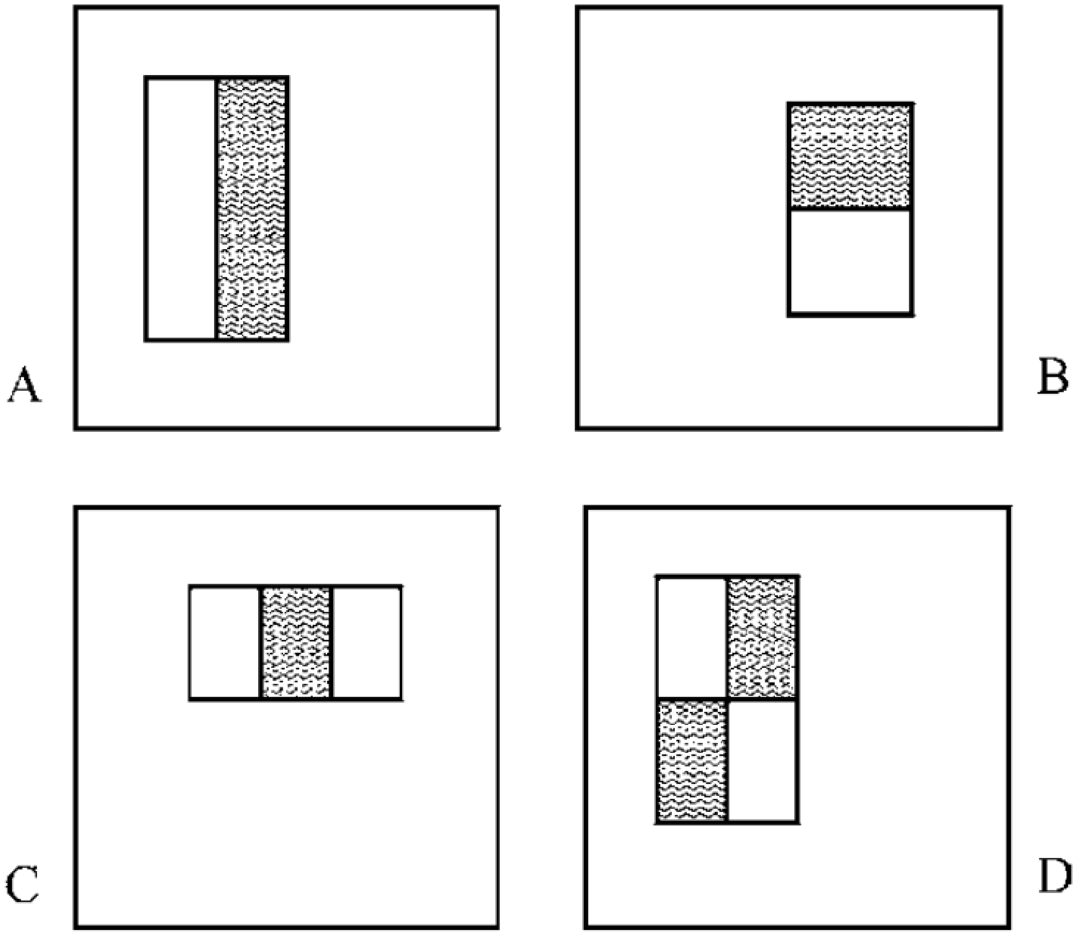

Figure 1: Haar-Like Features from [VJ04]. Chosen features are A and B.

1.1.5 Function: adaBoost

This function governs the construction of classifiers that fit to a particular false positive rate, by iteratively boosting selected haar-like features. The algorithm proceeds as follows: Of note is the use of the threshold value: $\frac{1}{2}\sum_i \alpha_i$. This is drawn explicitly from [VJ04], as they describe the threshold function. It is in terms of this value that I define Θ, a value which is calculated with the setBigTheta function described later on.

1.1.6 Function: setPolarityThreshold

This function implements a component of the *bestLearner* function, and is effectively a helper function. For a given feature, this function first evaluates the feature across all images and sorts the resulting values. Next, the sums T^-, T^+, S^-, S^+ are calculated, and design decisions

Algorithm 2 AdaBoost Implementation

```
• Initialize weights $w = \{\frac{1}{2m}, \frac{1}{2l} | m = \text{ num faces } l = \text{ num backgrounds}\}$
• While $t < T$ :
    – Normalize weights: $w = \frac{w}{\sum_i w_i}$
    – Set $learner = bestLearner(w)$
    – Set $\epsilon$ = error corresponding to *learner*
    – Set $\alpha = \log \frac{1-\epsilon}{\epsilon} = \log \frac{1}{\beta}$
    – Update weights $w_i = w_i \times \beta^{1-e_i}$ where $e_i = 1$ if example $i$ is classified correctly
      by *learner*, 0 otherwise
    – Append *learner* and $\alpha$ to classifier set
    – Run *setBigTheta* to set $\Theta$ such that the current hypothesis does not have any
      false negatives
    – Determine false positive rate $fp$, and if $fp > 0.3$, add a boosting round
• Return classifier set and $\Theta$, defined and used later as:
```

$$C = \begin{cases} 1 \text{ if } \sum_i^T \alpha_i h_i(x) + \Theta \geq \frac{1}{2} \sum_i^T \alpha_i \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

were made here to greatly optimize performance.

First, $T^{\pm}$ are sums of the weights of all of the positive examples and negative examples respectively. As such, these sums change only with updates to the weights themselves, so these sums are calculated a single time for each different set of weights.

Additionally, two new lists are maintained: SS_{-} and SSP. These lists are unique to each set of weights and each evaluated feature, and therefore have to be constructed once for each feature. The i-th entry of SS_{-} and SSP is a cumulative sum of all of the weights of negative or positive examples with indices $j < i$. Now to evaluate:

$$e = \min\{S^{+} + (T^{-} - S^{-}), S^{-} + (T^{+} - S^{+})\} \tag{2}$$

Each particular value of S^{+}, S^{-} can be found quickly by indexing into the SSP, SS_{-} array. As a result, this calculation is greatly optimized. This information is used to determine the threshold θ and the polarity p corresponding to the side of the minimum.

1.1.7 Function: bestLearner

This function chooses the best weak classifier in accordance with the specification in [VJ04] page 6. More specifically, each feature is first run through the *setPolarityThreshold* function that chooses a p and θ value. With these values, ϵ is calculated, corresponding to:

$$\epsilon = \sum_{i} w_i |h(x_i, f, p, \theta) - y_i| \tag{3}$$

After which, the minimum ϵ and corresponding h are chosen and returned, along with a calculation of the beta value: $\beta = \frac{\epsilon}{1-\epsilon}$

1.1.8 Function: setBigTheta

This function sets the value of Θ corresponding to each classifier such that the classifier produces no false negatives on the set of training examples. To do this, each classifier is evaluated on each image, and if the prediction is a background but the image is a face, we have:

$$\sum_{t}^{T} \alpha_t h_t(x) < \frac{1}{2}\sum_{t}^{T} \alpha_t \tag{4}$$

To correct for this, we can set $\Theta = \frac{1}{2}\sum_{t}^{T} \alpha_t - \sum_{t}^{T} \alpha_t h_t(x)$, so that this evaluation becomes:

$$\sum_{t}^{T} \alpha_t h_t(x) + \Theta < \frac{1}{2}\sum_{t}^{T} \alpha_t$$
$$\sum_{t}^{T} \alpha_t h_t(x) + \frac{1}{2}\sum_{t}^{T} \alpha_t - \sum_{t}^{T} \alpha_t h_t(x) = \frac{1}{2}\sum_{t}^{T} \alpha_t$$

And thus this classifier will return a prediction of a face for this example.

This procedure is continued across all weak learners comprising a classifier, with $\Theta = \max\{\Theta, \frac{1}{2}\sum_t^T \alpha_t - \sum_t^T \alpha_t h_t(x)\}$.

The final result of this procedure is that each weighted weak learner produces no false negatives on the example set.

1.1.9 Function: classifierCascade

This function governs the evaluation of a testing image through a classifier cascade. Simply, each classifier stage is used to predict faces from all sub-windows of the testing image, and predicted backgrounds are discarded from the set. The next stage of the cascade uses the newly reduced set of sub-windows that were predicted to be faces from the previous stage, and makes its own predictions, again discarding non-faces. This is continued until all stages are exhausted.

2 Results

2.1 Cascade Results

In this implementation, a full production run training on all 4000 training images and testing on the provided "class.jpg" picture showed these important characteristics:

In the full training set example 3, a classifier cascade of 4 stages was used, comprised of 9, 23, 6, and 6 weak classifiers, respectively. The error rates of each stage of the classifier are 29.5%, 29.1%, 28.6%, and 28%, respectively as well.

The time to train each classifier is somewhat long, with the entire training set requiring approximately 18 hours to complete. I suspect there to be implementation dependent optimizations that could have saved time. In particular, my implementation chooses to train many different features (strides of 8 pixels, with two feature types and varying dimensions of features). The overall complexity of these features may not contribute heavily to accuracy of the result, and significantly adds computation complexity, which indicates that a smaller feature set could potentially have been used.

In image 2, a small subset of the training images were used, and an exclusion rule was enforced to select one out of any overlapping set of predicted windows. The exclusion rule prefers windows that are closer to the upper left of the test image, not based upon feature value.

In image 3, the full training set of images were used and the exclusion rule referenced above is not used, as it was added later. As is evident, there are many false positives still present in the image.

3 Discussion

The results of this algorithm show that adaBoost may not be sufficient for face detection accuracy to appropriate levels for the task. It may be that implementing more feature types

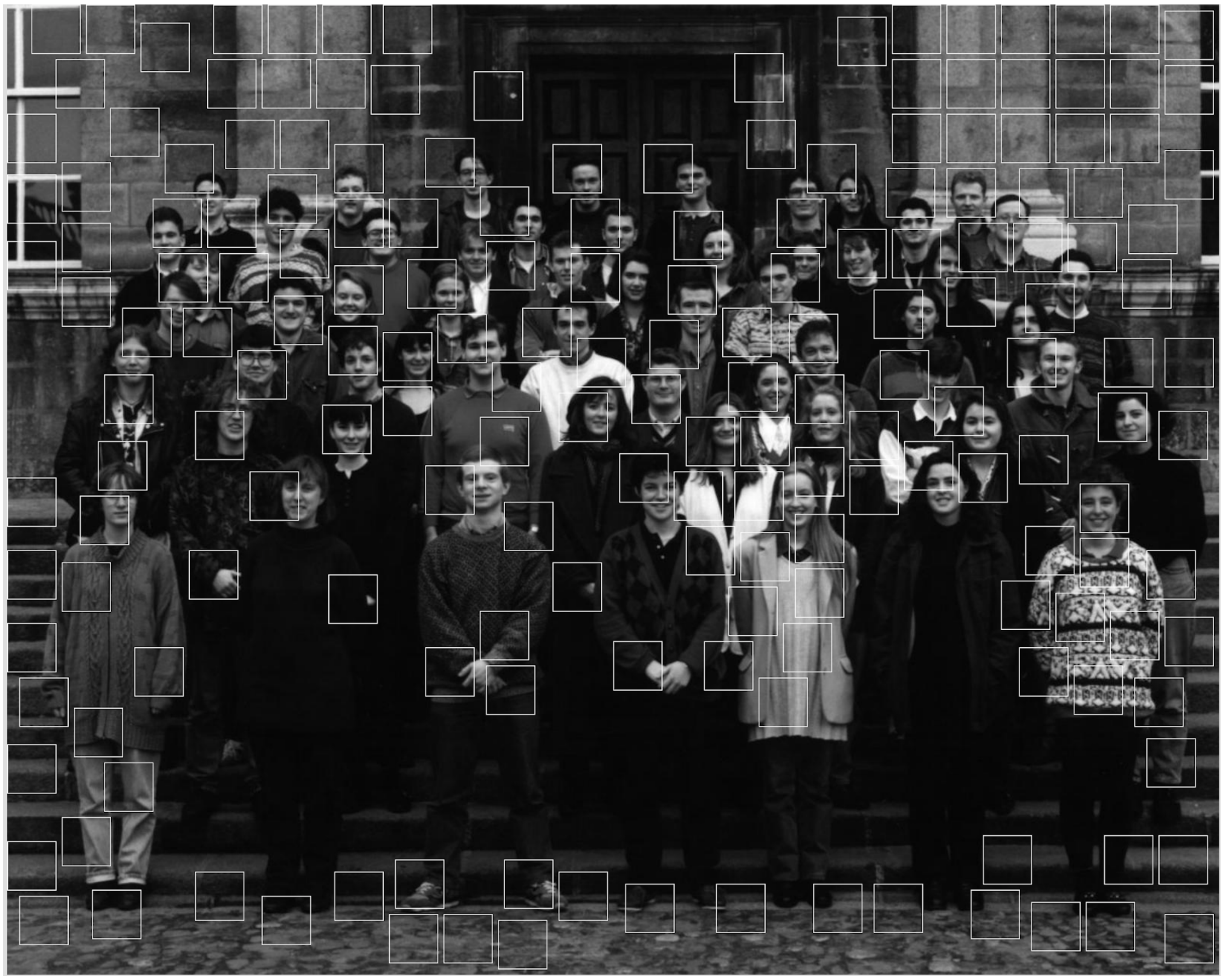

Figure 2: Example run of Viola Jones Implementation, on small subset of training images

could provide greater accuracy, i.e expanding beyond the simple two-rectangle features. Results were more accurate when the false positive rate threshold for each classifier stage was reduced. Specifically, false positive rate thresholds of 0.03 and 0.003 gave more accurate results without as many classifier cascade stages. This may be because of the specific implementation details surrounding the classifier cascade. In the future, optimizations surrounding feature type and selection would be where I would begin to optimize this implementation. Additionally, the false positive threshold could be another point to examine and optimize.

References

[VJ04] Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

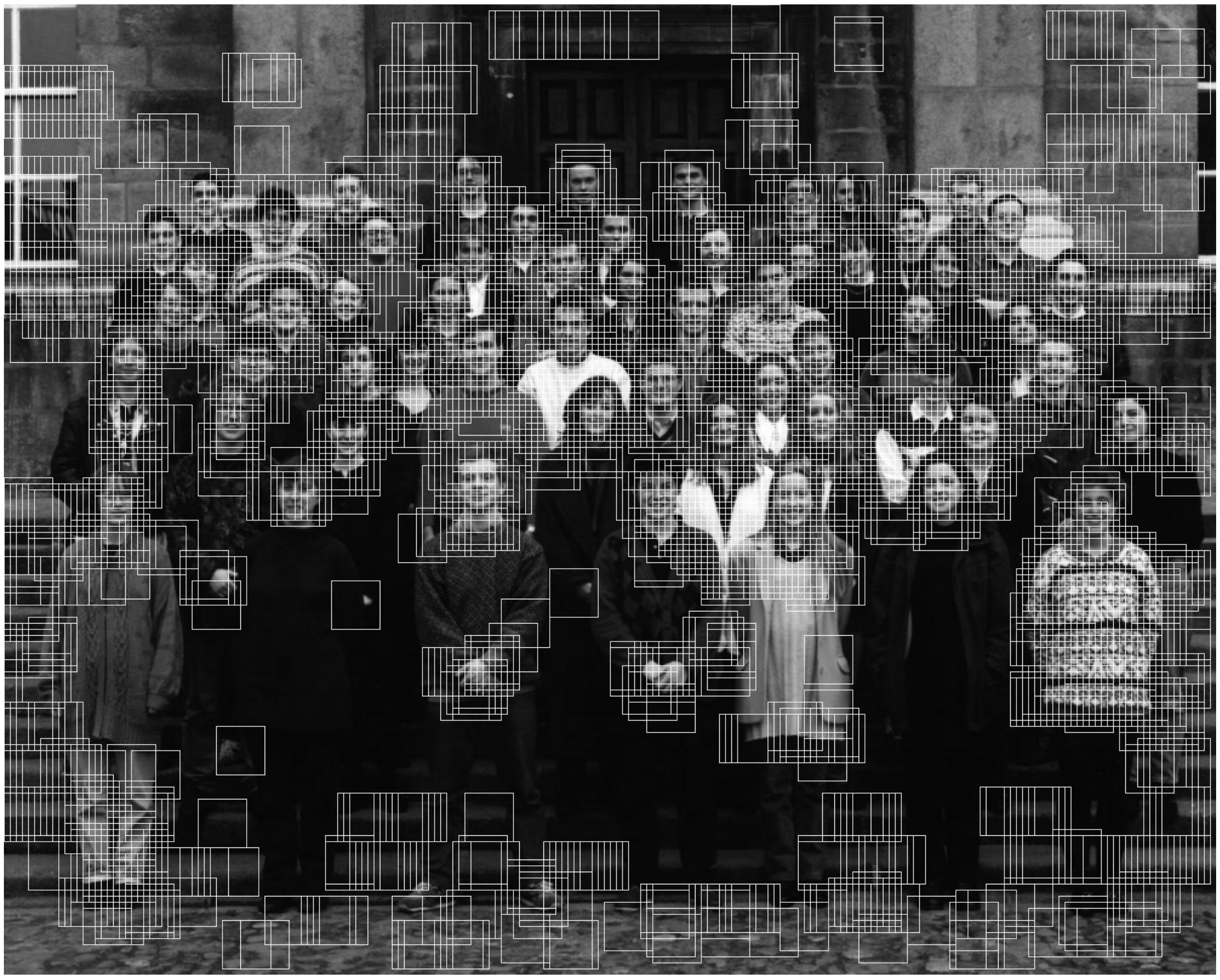

Figure 3: Full training run of Viola Jones Implementation without exclusion rule