CMSC 254
Machine Learning
Homework 4

Casper Neo

14 February 2017

I implemented the Viola-Jones approach to facial recognition of using a cascade of adaboosted haar functions to quickly identify and discard backgrounds and identify faces.

Training the cascade on the data yields a sequence of functions, which all have to return true for an image sub-window to be considered a face. Each function f_i is the output of the adaboost algorithm run over a number of Haar functions n_i and dataset D_i until the false positive rate (number of backgrounds mistaken for faces) fall below a threshold τ_i. The data set D_i consists of all the faces in the training data and all the false positive backgrounds that passed the cascade until round i. My parameters are the number of Haar functions, the false positive rate threshold, and the length of the cascade m; summarized by sequence $\{(n_i, \tau_i\}_{i=1}^{m}$.

In each boosting round, I set the false negative rate to be zero in the training data set (which inflates the false positive rate).

I chose to use the following Whole Window Haar functions. The first 127 were considered simple and the latter were considered complex. The simple and complex functions where shuffled within each group so when the cascade picks a number $n > 127$ of weak learners, it is ensured to pick all the simple functions and a random sample of complex functions, or if $n \leq 127$ just a random sample of weak learners.

Haar function	number of functions
1*1	1
2*1	63
1*2	63
3*1	$\binom{63}{2} = 1953$
1*3	$\binom{63}{2} = 1953$
2*2	$63^2 = 3969$
total	8002

The program ran quite slowly (over 4 hours to train on 4000 images) so I implemented a parallelized version which stored Haar features differently. In the original implementation each Haar feature was given its own lambda function, in the parallelized implementation, I kept a dictionary of functions holding each type of Haar function $1 * 1, 2 * 1, etc$, and computed features $f = (\text{Haar type}, args)$ where args specified the boundaries in the Haar function. I did this because python did not easily support pickling of functions while pickling arguments is required for mapping to multiple threads. Because cProfile fails on multithreaded programs I'm not actually sure if my multithreaded program runs significantly faster than the original. But I think so.

I found initially that constructing the cascade played a significant role in the performance of my algorithm. The longer the cascade, and stricter the FDR threshold, the fewer background images are available for the latter functions which then tend to overfit. However once I realized that the Haar features (about figure 7) I was supposed to implement should act on sub-windows rather than over the whole detector, performance improved

dramatically. By having more and more diverse features, I cut down the number of boosting selections in each round of the cascade by roughly a factor of 10. This was more significant than multithreading my program which only increased speed by a factor of 4-8. After threading, calculating 8000 HAAR features over 4000 images takes about 50 seconds. When the feature space is rich enough that you only need between 12 and 30 selections per cascade, it runs pretty fast.

The following images the given "class.jpg" with white squares drawn around faces along with the cascade specification and data provided. One important design choice that differs from the paper, which I realized late into the project, is that my decision stumps return from $\{1, -1\}$ rather than from $\{1, 0\}$. This is definitely a very significant difference as every component function in the boosted function always has an opinion, but I do not have particularly good intuition as to how that would manifest beyond longer times to convergence.

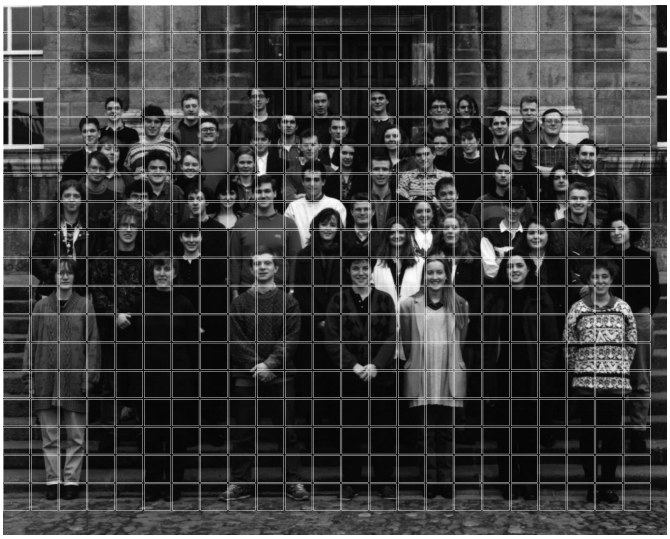

Figure 1: Data: 100 images and 100 backgrounds Cascade = $[(64, .5)]$

Clearly I did not give the algorithm much data and set the false positive rate too high in the cascade of 1. The algorithm identifies every square as a face. Note that I implemented a "face exclusion" rule, and I process the picture in two pixel steps starting from the top left.

Figure 2: Data: 2000 images and 2000 backgrounds Cascade = $[(127, .5), (500, .5), (1000, .3), (2000, .3)]$

When I ran the algorithm to these specifications I found that it took hundreds of rounds of boosting in each step of the cascade suggesting that it would become over fit. Nonetheless the algorithm is doing a decent job at capturing faces despite a high false positive rate. It was this output that let me notice the following trade off, the lower the FPR the more rounds of boosting is required to reach the FDR rate and that boosted function will become more over fit. But if you increase the FDR rate and increase the length of the cascade, the later functions will become overfit to the data because there will be so few background images.

Figure 3: Data: 2000 images and 2000 backgrounds Cascade = $[(127, .3), (127, .3), (127, .3)]$

For this image I used the specifications recommended in the assignment write up. Only the most basic Haar functions and three rounds of boosting until $FPR < .3$. Clearly this does about as well as the longer chain with more functions in the previous cascade. Considering how few functions were used in this cascade this set up is strictly superior.

Figure 4: Data: 300 images and 300 backgrounds Cascade = $[(127, .5), (500, .5), (1000, .5), (2000, .5), (5000, .3)]$

This cascade does about as well as the previous ones. Except it does so with 15% the data. That is remarkable. I suspect setting a higher threshold for the simplest 127 functions allowed the program to focus on filtering the easiest backgrounds without overfitting. This trained with 42, 63, 48, 24, and 6 adaboost selections for the respective rounds in the cascade. I believe the low number of features selected in each round of boosting reflects that the algorithm was not very overfit.

Figure 5: Data: 2000 images and 2000 backgrounds Cascade = $[(127, .5), (500, .5), (1000, .5), (2000, .5), (5000, .3)]$

Training the same cascade on even more data does not yield significantly better results. Each round took 161, 343, 423, 323, 59 selectors. While more selectors may be expected for more data, it seems that we've run into the overfitting problem again.

Figure 6: Data: 2000 images and 2000 backgrounds Cascade = $[(8002, .3), (8002, .3), (8002, .3)]$

Again I tried the specifications in the write up but with all my haar functions. This does worse than the original specifications probably due to overfitting. Given there are 8002 functions and only 4000 images (at maximum since we discard images between cascade rounds) it is very easy to overfit. Additionally since I make Haar features for every window there is probably too much reliance on the difference between single pixels.

Figure 7: Data: 2000 images and 2000 backgrounds Cascade = $[(500, .3), (1000, .3), (2000, .3)]$

This takes 90, 110, and 78 boosting selections each. I changed the set up such that the stride length is two. Does not seem to do much better.

It was a this point where I realized my Haar functions split the whole detector into the various columns and rows rather than looking at a subwindow. I adjusted my haar functions to check boxes of length and height 2-10 starting at odd numbered pixels. I only used 1x1s, 2x1s, and 2x2s. In total this made 243675 functions. Again I shuffle my functions before using them.

Figure 8: Data: 100 images and 100 backgrounds Cascade = $[(500, .3), (1000, .3), (2000, .3)]$

This output despite being over 100 images seems strictly superior to my previous results. It seems having better haar functions is incredibly useful.

Figure 9: Data: 2000 images and 2000 backgrounds Cascade = $[(1000, .3), (2000, .3), (3000, .3)]$

I coded up 1x3, 3x1, and 2x2 sub-window functions where the splits are equally spaced let them range in both height and width from 3 to 40 pixels since in the test image, faces seem to cover half to 2 thirds of the detector window. In total there are now 15,649,350 potential features that I'm randomly sampling from. However this doesn't do that great.

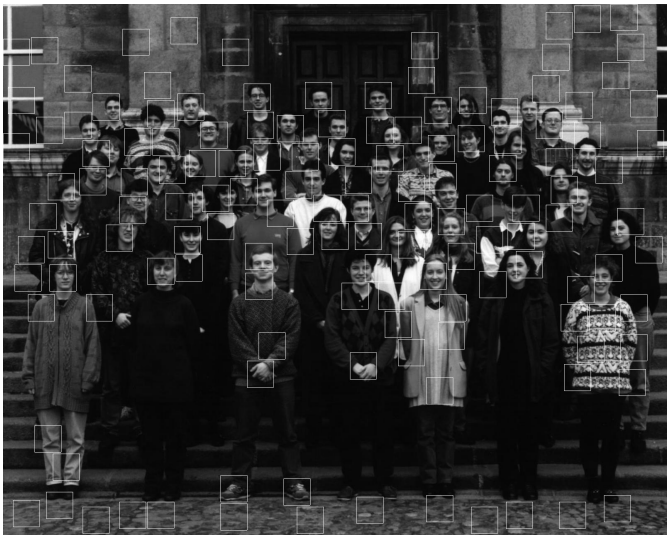

Figure 10: Data: 2000 images and 2000 backgrounds Cascade = $[(2000, .3), (4000, .3), (8000, .3)]$

I'm still finding a lot of false positives. This one trained on the 1x2s, 2x1s, 1x3s, and 2x2s.

Figure 11: Data: 2000 images and 2000 backgrounds Cascade = $[(2000, .3), (4000, .3), (8000, .3)]$

I read Professor Kondor's piazza post of ignoring the threshold in the final output of the cascade and that made a huge difference in false positive rate. The false negative rate is a little higher but this output is better.

Figure 12: Data: 2000 images and 2000 backgrounds Cascade = $[(2000, .5), (4000, .5), (8000, .5), (16000, .3)]$

This time I added more print statements to my output and found that over half of the selected features are 2x2 sub-windows. The next most common seem to be 1x3 followed by 3x1. I suspect it has to do with contrast and edge detection. The cascade here selected from 1x1, 2x1, 1x2, 1x3, 3x1, 2x2 features with window size from 10-64 pixels. I decided that allowing the sub-window sizes to be too small would make the program more likely to overfit as it can find the common pixel that all backgrounds have in common. I wouldn't say adjusting the cascade significantly improved the results.