**Assignment 4 – Writeup**

I will explain some of my design choices by explaining the sequence of functions that should be run from face.py

First, I had a main directory that includes face.py (code to run the algorithm) and class.jpg

The training examples were unzipped in two sub-directories, “backgrounds” and “faces”

All of the preliminary data preparation are stored in the “prep” subdirectory

**grey = get_grey(“class.img”)**

**classimg = get_iimages_for_class(grey)**

This converts the img to a greyscale, and builds an integral image representation

**p0_100, c0_100 = get_patches_for_class(classimg, 2, 64, 0, 98)**

**np.save(“prep/p0_100”, p0_100)**

**np.save(“prep/c0_100”, c0_100)**

**p100_200, c100_200 = get_patches_for_class(classimg, 2, 64, 100, 198)**

**… all the way up to**

**P1100_, c1100_ = get_patches_for_class(classimg, 2, 64, 1100, 1260)**

**np.save(“prep/p1100_”, p1100_)**

**np.save(“prep/c1100”, c1100_)**

I divided the image into 12 rows and saved the integral image representations separately, because the entire process took a long time. Ps are the patches (1*4096 arrays), and Cs are the coordinates corresponding to the upper left point of a 64*64 patch.

**path = 'C:\\Users\\gwkim\\Desktop\\Courses\\Winter 2017\\ML\\hw4”**

**get_iimages(path+”\\faces”, path+”\\backgrounds”, “prep/trainarray”)**

**examples = np.load(“prep/trainarray.npy”)**

**labels = [1]*2000 + [-1]*2000**

This is a preparation stage for building the integral image representation of the 4000 training images combined.

Labels are set to 2000 1s and 2000 -1s, because I put the 2000 face images first.

Design choice: Even after improving speed significantly, training a model with roughly 1500 features and full 4000 examples, with 0.3 target false positive rate for each cascade and 0.01 final target false positive rate, took about 60~90 minutes.

In this case, each iteration in first cascade took about 70 seconds, second cascade took 40 seconds, third cascade 35, and fourth cascade 33. The iteration duration reduced because I excluded correct negative examples after each iteration. The iteration number per each cascade, on the other hand, increased. (roughly 9 -> 22 -> 30 -> 34. 4 cascades are always guaranteed because 0.3^4 < 0.01)

I forgot to record the training error, but I recall that it was always at around 0.1~0.2 per each cascade-iteration pair.

I could not risk adding too many features at one training cycle and not being able to train a model in time, so my thought was to train multiple models with different-sized features, and then only classify a 64*64 patch as a face if it is classified as a face in all of the models. This is based on the idea that I set a "Theta" so that I don't get any false negatives on the training data, while being a little more lenient on the false positives (0.01 final target). Thus, I thought I should exclude some false positives by checking with other models.

However, I was worried that a face may not pass all of the classifier models if I introduce too many models, so I only picked a few that showed the least false positive rate. I just tested each model on p0_100 (patches of class.jpg that had height 0~100), which did not have any face, and checked which one shows the least patches classified as faces.

The two models I eventually ended up using are the following:

**ftbl1 = get_featuretbl(2, 3, 5, 6, 8, 2)**

**haaat_list1 = cascade_adaboost(examples, labels, ftbl1, 0.3, 0.01)**

**ftbl2 = get_featuretbl(2, 5, 5, 10, 10, 1)**

**haaat_list2 = cascade_adaboost(examples, labels, ftbl2, 0.3, 0.01)**

Because I sometimes turned off my ipython console, I saved the parameters for the resulting model, and used function "reconstruct_haaat_list" to get the model back.

**haaat_list1 = reconstruct_haaat_list(np.load("prep/model1_hargs.npy"), np.load("prep/model1_alphas.npy"), get_featuretbl(2, 3, 5, 6, 8, 2))**

```
haaat_list2 = reconstruct_haaat_list(np.load("prep/model2_hargs.npy"),
np.load("prep/model2_alphas.npy"), get_featuretbl(2, 5, 5, 10, 10, 1))
```

```
p0_100 = np.load('prep/p0_100.npy')
p100_200 = np.load('prep/p100_200.npy')
```

**all the way to**

```
p1000_1100 = np.load('prep/p1000_1100.npy')
p1100_ = np.load('prep/p1100_.npy')
```

```
c0_100 = np.load("prep/c0_100.npy")
c100_200 = np.load("prep/c100_200.npy")
```

**all the way to**

```
c1000_1100 = np.load('prep/c1000_1100.npy')
c1100_ = np.load('prep/c1100_.npy')
```

Just loading the patches and coordinates that were saved before.

```
f01_1 = get_faces_0(haaat_list1, p0_100, c0_100)
f01_2 = get_faces_0(haaat_list2, p0_100, c0_100)
f12_1 = get_faces_0(haaat_list1, p100_200, c100_200)
f12_2 = get_faces_0(haaat_list2, p100_200, c100_200)
```

**all the way to**

```
f1011_1 = get_faces_0(haaat_list1, p1000_1100, c1000_1100)
f1011_2 = get_faces_0(haaat_list2, p1000_1100, c1000_1100)
f11_1 = get_faces_0(haaat_list1, p1100_, c1100_)
f11_2 = get_faces_0(haaat_list2, p1100_, c1100_)
```

for each combination of patches and coordinates (12 rows in class.jpg), and for the two models I am using, I am receiving the list of upper-left coordinates, whose patches are classified as faces by each model.

```
f01 = common_list_all([f01_1, f01_2])

f12 = common_list_all([f12_1, f12_2])
```

**all the way to**

```
f1011 = common_list_all([f1011_1, f1011_2])

f11 = common_list_all([f11_1, f11_2])
```

As mentioned before, I am only counting the patches that are classified as faces in both models.

```
f = f01+f12+f23+f34+f45+f56+f67+f78+f89+f910+f1011+f11

final_f, final_nf, coord = pick_one_face_and_filter(f, 30*math.sqrt(2), 7)

draw_result(final_f, "final.png")
```

After combining the face-patches of all rows, I am running the "pick_one_face_and_filter" function to apply the exclusion rule for many overlapping face-patches.

At first I have grouped the points in the list "f". If the newly inspected point is not apart from the existing groups (averaged location) by more than a gap of 30*sqrt(2), I included the new point into the corresponding group. After doing that, I rounded the averaged location so that each group's average is a valid upper-left coordinate for one of the patches.

The filter_threshold of 7 is a new term I introduced after having found that there are still many false positives. This is probably because I trained the model with limited time. My guess is that if I set the target final false positive rate to a lot less than 0.01, less false positive detections would have occurred.

My solution was to exclude a group of patches, if the size of the group is less than 7. This is because I observed that many false positives were only detected in a few adjacent patches, while the true face should be detected by many. I tested with a few threshold values and concluded that a value of 7 is the most appropriate in trying to get the most faces correct, while limiting the emergence of false positives.

*If the model cannot be perfectly accurate, depending on what the criteria is (getting all true positives vs. avoiding as many false positives as possible), the threshold can be adjusted. For example, if the threshold is lowered to 4, the code would capture all faces, although the false positives also increase.

The output is stored in final.png, where each detected patch is indicated with white squares.