CMSC25400 Homework 4

University of Chicago, Fall 2017 — due Tuesday, February 12 at 11:59pm

The files `faces.tar.gz` and `background.tar.gz` contain 2000 images of faces and non-face image patches, respectively. Your task in this assignment is to implement a Viola–Jones type boosting based detector cascade, as described in the lectures, to distinguish between these two classes, and hence detect faces in larger images. We provided a standard test image to test your algorithm. To detect the faces in the image, use a sliding window to evaluate your trained detector on each 64×64 patch. Report your results either by giving the coordinates of each patch where you detected a face, or, better yet, by drawing a square around the patch. You might find that you get many overlapping detections around each face. To avoid this problem use an exclusion rule to avoid overlaps. Please include a detailed writeup in your submission explaining any design choices that you have made, how many classifiers you have in your cascade, what the training error of each classifier is, and how long it took to train each stage of the cascade.

You have a lot of freedom in how exactly to implement the algorithm. However, if you're having trouble getting started, we suggest implementing the following functions. These are merely suggestions and you should feel free to deviate from our prescribed steps if you think you have a more convenient approach. These function signatures are written in Python but you may choose to use whatever language you're most comfortable with.

1. Write a function to load all the data into a matrix

```
def load_data(faces_dir, background_dir):
    '''
    faces_dir: string, file location of the faces directory
    background_dir: string, file location of the background directory
    Returns: a tuple of numpy matrices
        - a matrix of size N x 64 x 64 of the images
        - a matrix of size N x 1 of the class labels(+1 for a face, -1 for background)
    '''
```

   You may find it helpful to use library functions from scipy or imageio to load the jpeg files into a numpy matrix. Note that the images are in RGB format so once you load them, each image will be a $64 \times 64 \times 3$ matrix. We won't need the color, so convert the image to greyscale. You can do this by averaging each pixel over the color index(last index of the matrix).

2. Recall that all of the Haar filters can be computed very quickly if you already have the integral image representation pre-computed. Implement a function that computes this integral image representation. The output matrix should satisfy:

$$Output_{i,x,y} = \sum_{a=1}^{x} \sum_{b=1}^{y} Input_{i,a,b}, \text{ for all images } i,\ 1 \leq a, b \leq 64$$

```
def compute_integral_image(imgs):
    '''
    imgs: numpy matrix of size N x 64 x 64, where N = total number of images
    Returns: a matrix of size N x 64 x 64 of the integral image reprsentation
    '''
```

3. Construct a list that will hold the relevant information for each Har-filter feature. In the boosting slides we showed a whole assortment of Haar filters that you could potentially use, but for this assignment just the 2-rectangles will be sufficient. A 2-rectangle feature is uniquely determined by:

(a) pixel location (x_s, y_s) of the upper left corner of the shaded rectangle
(b) pixel location (x'_s, y'_s) of the bottom right corner of the shaded rectangle
(c) pixel location (x_u, y_u) of the upper left corner of the unshaded rectangle
(d) pixel location (x'_u, y'_u) of the bottom right corner of the unshaded rectangle

Remember that once you have the integral image reprsentation computed, we just need the pixel locations of the relevant corners of this filter to compute the corresponding feature on an image.

```
def feature_list():
    '''
    Returns: list of relevant pixel locations for each feature.
        The ith index of this list should be a list of the four relevant
        pixel locations needed to compute the ith feature
    '''
```

4. Implement a function that computes the features. Given a feature index (an index into the datastructure computed in the previous step), you have the necessary information to evaluate this feature on a given image.

```
def compute_feature(int_img_rep, feat_lst, feat_idx):
    '''
    int_img_rep: the N x 64 x 64 numpy matrix of the integral image representation
    feat_lst: list of features
    feat_idx: integer, index of a feature (in feat_lst)
    Returns: an N x 1 numpy matrix of the feature evaluations for each image
    '''
```

5. At each iteration of AdaBoost, we pick the best weak learner that minimizes the weighted training error. Here, our weak learners are given by $\{h_{i,p,\theta}\}$ where i is a feature index, $p \in \{+1, -1\}$, and $\theta \in \mathbb{R}$. In lecture, we saw an efficient way of computing the optimal p and θ values for for a fixed feature index i (see slide 27/31 of the boosting slides). Implement a function that performs this computation.

```
def opt_p_theta(int_img_rep, feat_lst, weights, feat_idx):
    '''
    int_img_rep: the N x 64 x 64 numpy matrix of the integral image representation
    feat_lst: list of features
    weights: an N x 1 matrix containing the weights for each datapoint
    feat_idx: integer, index of the feature to compute on all of the images
    Returns: the optimal p and theta values for the given feat_idx
    '''
```

Implement a function that computes the predictions of a given weak learner:

```
def eval_learner(int_img_rep, feat_lst, feat_idx, p, theta):
    '''
    int_img_rep: the N x 64 x 64 numpy matrix of the integral image representation
    feat_lst: list of features
    feat_idx: integer, index of the feature for this weak learner
    p: +1 or -1, polarity
    theta: float, threshold
    Returns: N x 1 vector of label predictions for the given weak learner
    '''
```

Write a function to compute the weighted error rate of a given weak learner:

```
def error_rate(int_img_rep, feat_lst, weights, feat_idx, p, theta):
    '''
    int_img_rep: the N x 64 x 64 numpy matrix of the integral image representation
    feat_lst: list of features
    weights: an N x 1 matrix containing the weights for each datapoint
    feat_idx: integer, index of the feature for this weak learner
    p: +1 or -1, polarity
    theta: float, threshold
    Returns: the weighted error rate of this weak learner
    '''
```

Finally, implement a function that finds the optimal weak learner $h_{i,p,\theta}$ over the training data. This function should be fairly simple given the previous functions:

```
def opt_weaklearner(int_img_rep, weights, feat_lst):
    '''
    int_img_rep: the N x 64 x 64 numpy matrix of the integral image representation
    weights: N x 1 matrix of weights of each datapoint
    feat_lst: list of features
    Returns: the i, p, and theta values for the optimal weak learner
    '''
```

6. Implement a function to update the weights of each datapoint. Recall that the update rule is given by:

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i)\exp(-\alpha_t y_i, h_t(x_i))}{Z_t} \\ \alpha_t &= \frac{1}{2} ln \frac{1-\epsilon_t}{\epsilon_t} \\ Z_t &= 2\sqrt{\epsilon_t(1-\epsilon_t)} \end{aligned}$$

```
def update_weights(weights, error_rate, y_pred, y_true):
    '''
    weights: N x 1 matrix containing the weights of each datapoint
    error_rate: the weighted error rate
    y_pred: N x 1 matrix of predicted labels
    y_true: N x 1 matrix of true labels
    Returns: N x 1 matrix of the updated weights
    '''
```

At this point you should have most of the helper functions you'd want to perform the main components of the AdaBoost algorithm. Remember, this is just a set of guidelines. It is not an exhaustive list of the functions you'll need to have a working implementation of the Viola Jones face detector.

In addition, here are a few tips:

1. Use the Haar-like features described in the Viola–Jones paper and simple decision stumps (with varying threshold) as weak learners. Each round of boosting will select a single (i, p, θ) combination (where i indexes the feature, $p \in \{-1, +1\}$ is the polarity, and θ is the threshold) to add to your classifier.
2. The original Viola–Jones paper used three distinct types of Haar–like features. For the purposes of this assignment using just the simplest "two-rectangle" features is probably sufficient. However, for extra credit you might implement the other features as well.

3. The hypothesis returned by regular AdaBoost is $h(x) = \text{sgn}\big(\sum_{t=1}^{T} \alpha_t h_t(x)\big)$. However, in a classifier cascade it is critical that each classifier have a low false negative rate. Therefore, we suggest that you instead use $h(x) = \text{sgn}\big(\sum_{t=1}^{T} \alpha_t h_t(x) - \Theta\big)$, where Θ is set so that there are no missed detections (false negatives) on your training date at all.
4. A single booster with such an artificially low false negative rate will have a fairly high false positive rate. We suggest that you keep adding new features until the false positive rate falls below about 30%. You will probably find that less than a dozen rounds a boosting are sufficient for this. Chaining several such boosters together will push the false detection rate of the entire cascade down to acceptable levels.
5. One key idea in the Viola–Jones paper is that it is feasible to have a large number of base learners (Haar features) as long as the base learners are very fast to compute. These fetures should therefore always be computed on the fly (from the integral image representations) rather than computed once and then stored.
6. Speed might nonetheless be an issue in your implementation. We suggest that in the development stage you work with smaller data, and then scale up to the entire dataset on the final "production run". In addition, don't be afraid to (a) reduce the number of features by using a stride of size 2 or 4 as opposed to trying every Haar-like feature in every possible pixel location (b) reduce the training set, by training on just 1000 or even less images from each class. Try and write code that is relative efficient — that can also make a big difference.
7. Depending on the efficiency of your implementation you might want to start with smaller number of features than currently used in the provided code. One way to achieve that is to use Haar features of larger size (i.e. window size larger than 2×1 and 1×2) and/or larger strides.