Problem Set 6: Image Classification (Faces)

Description

In this problem set you will be implementing face recognition using PCA, (Ada)Boosting, and the Viola-Jones algorithm.

Learning Objectives

- Learn the advantages and disadvantages of PCA, Boosting, and Viola-Jones.
- Learn how face detection/recognition work, explore the pros & cons of various techniques applied to this problem area.
- Identify the inherent challenges of working on these face detection methods.

Problem Overview

Methods to be used: In this assignment you will be implementing PCA, Boosting, and HaarFeatures algorithms from scratch. Unlike previous problem sets, you will be coding them without using OpenCV functions dedicated to solve the problem.

RULES: You may use image processing functions to find color channels, load images, find edges (such as with Canny). Don't forget that those have a variety of parameters and you may need to experiment with them. There are certain functions that may not be allowed and are specified in the assignment's autograder Piazza post.

Refer to this problem set's autograder post for a list of banned function calls.

Please do not use absolute paths in your submission code. All paths should be relative to the submission directory. Any submissions with absolute paths are in danger of receiving a penalty!

Obtaining the Starter Files:

```
Obtain the starter code from canvas under files.
```

Programming Instructions

Your main programming task is to complete the api described in the file **ps6.py**. The driver program **experiment.py** helps to illustrate the intended use and will output images to verify your results. Additionally there is a file **ps6_test.py** that will test your implementation.

Write-up Instructions

Create **ps6_report.pdf** - a PDF file that shows all your output for the problem set, including images labeled appropriately (by filename, e.g. ps6-1-a-1.png) so it is clear which section they are for and the small number of written responses necessary to answer some of the questions (as indicated). For a guide as to how to showcase your results, please refer to the powerpoint template for PS6.

How to submit:

Two assignments have been created on Gradescope: one for the report - **PS6_report**, and the other for the code - **PS6_code**.

- **Report:** the report (PDF only) must be submitted to the **PS6_report** assignment.
- **Code: all files** must be submitted to the **PS6_code** assignment. **DO NOT** upload zipped folders or any sub-folders, please **upload each file individually**. Drag and drop all files into Gradescope.

Notes:

- You can only submit to the autograder **10** times in an hour. You'll receive a message like "You have exceeded the number of submissions in the last hour. Please wait for 36.0 mins before you submit again." when you exceed those 10 submissions. You'll also receive a message "You can submit 8 times in the next 53.0 mins" with each submission so that you may keep track of your submissions.
- If you wish to modify the autograder functions, create a copy of those functions and **DO NOT** mess with the original function call.

YOU MUST SUBMIT your report and code separately, i.e., two submissions for the code and the report, respectively. Only your last submission before the deadline will be counted for each of the code and the report.

Grading

The assignment will be graded out of 100 points. The last submission before the time limit will only be considered. The code portion (autograder) represents **50**% of the grade and the report the remaining **50**%.

The images included in your report must be generated using experiment.py. This file should be set to be run as is to verify your results. **Your report grade will be affected if we cannot reproduce your output images.**

The report grade breakdown is shown in the question heading. As for the code grade, you will be able to see it in the console message you receive when submitting.

Assignment Overview

1. PCA [30 points]

Principal component analysis (PCA) is a technique that converts a set of attributes into a smaller set of attributes, thus reducing dimensionality. Applying PCA to a dataset of face images generates eigenvalues and eigenvectors, in this context called eigenfaces. The generated eigenfaces can be used to represent any of the original faces. By only keeping the eigenfaces with the largest eigenvalues (and accordingly reducing the dimensionality) we can quickly perform face recognition.

In this part we will use PCA to identify people in face images. We will use the Yalefaces dataset, and will try to recognize each individual person.

a. **Loading images:** As part of learning how to process this type of input, you will complete the function `load_images`. Read each image in the images_files variable, resize it to the dimensions provided, and create a data structure X that contains all resized images. Each row of this array is a flattened image (see np.flatten).

You will also need the labels each image belongs to. Each image file has been named using the following format: *subject##.xyz.png*. We will use the number in ## as our label ("01" -> 1, "02" -> 2, etc.). Create a list of labels that match each image in X using the filename strings.

Next, use the X array to obtain the mean face (μ) by averaging each column. You will then use the resulting structure to reshape it to a 2D array and later save it as an image.

Code:
- `load_images(folder, size=(32, 32))`

Report:
- Mean face image: **ps6-1-a-1.png**

b. **PCA:** Now that we have the data points in X and the mean μ, we can go ahead and calculate the eigenvectors and eigenvalues to determine the vectors with largest covariance. See Eigenfaces for Recognition for more details. Using the equation from the lectures:

$$u^T \Sigma u \quad \text{where} \quad \Sigma = \frac{1}{M}\sum_{i=1}^{N} (x_i - \mu)(x_i - \mu)^T$$

You need to find the eigenvectors u. Luckily there is a function in Numpy `linalg.eigh` that can do this using Σ as an input.

Select the top 10 eigenvectors (in descending order) according to their eigenvalues and save them. These are our 10 eigenfaces.

Code:
- `pca(X, k)`

Report:
- Top 10 eigenfaces: **ps6-1-b-1.png**

c. **Face Recognition (classification):** Now that we have the PCA reduction method ready, let's continue with a simple (naive) classification method.

First, we need to split the data into a training and a test set. You will perform this operation in the `split_dataset` function. Next, the training stage is defined as obtaining the eigenvectors from using the training data. Each image face in the training and test set is then projected to the "face space" using these vectors.

Finally, find the eigenface in the training set that is closest to each projected face in the test set. We will

use the label that belongs to the training eigenface as our predicted class.

Your task here is to do a quick analysis of these results and include them in your report.

Code:
- `split_dataset(X, y, p)`

Report:
- Analyze the accuracy results over multiple iterations. Do these "predictions" perform better than randomly selecting a label between 1 and 15? Are there any changes in accuracy if you try low values of k? How about high values? Does this algorithm improve changing the split percentage p?

2. Boosting [20 points]

In this part we will classify face images based on the dataset's labeled gender [female or male]. You may find the contents on page 663 of Szeliski's (2010) book useful for this section (available on T-Square/Resources). Recall from the lectures that the idea behind boosting is to use a combination of "weak classifiers". Each weak classifier's vote is weighted based on its accuracy.

The Boosting class will contain the methods behind this algorithm. You will use the class `WeakClassifier` as a classifier ($h(x)$) which is implemented to predict based on threshold values. Boosting creates a classifier $H(x)$ which is the combination of simple weak classifiers.

Complete the `Boosting.train()` function with the Adaboost algorithm (modified for this problem set) :

Input: Positive (1) and negative (-1) training examples along with their labels

Initialize all the weights to $w_i \leftarrow \frac{1}{N}$, where $N$is the number of training examples.

For each training stage $j = 1 \ldots M$:

a) Renormalize the weights so they sum up to 1

b) Instantiate the weak classifier h with the training data and labels.
Train the classifier h
Get predictions $h(x)$ for all training examples $x \in X_{train}$

c) Find $\epsilon_j = \sum_i \quad w_i$ for weights where $h(x_i) \neq y_i$

d) Calculate $\alpha_j = \frac{1}{2} ln(\frac{1-\epsilon_j}{\varepsilon_j})$

e) If ϵ is greater than a (typically small) threshold:
Update the weights: $w_i \leftarrow w_i e^{-y_i \alpha_j h_j(x_i)}$
Else:
Stop the for loop

After obtaining a collection of weak classifiers, you can use them to predict a class label implementing the following equation in the `Boosting.predict()` function:

$$H(x) = sign[\sum_j \alpha_j h_j(x)]$$

Return an array of predictions the same length as the number of rows in X.

Additionally you will complete the `Boosting.evaluate()` function, which returns the number of correct and incorrect predictions after the training phase using X_{train} and y_{train} already stored in the class.

a. Using the Faces94 dataset, split the dataset into training (X_{train}) and testing (X_{test}) data with their respective labels (y_{train} and y_{test}). Perform the following tasks:

1. Establish a baseline to see how your classifier performs. Create predicted labels by selecting N random numbers $\in \{-1, 1\}$ where N is the number of training images. Report this method's accuracy (as a percentage): $100 * correct / total$.
2. Train `WeakClassifier` using the training data and report its accuracy percentage.
3. Train `Boosting.train()` for `num_iterations` and report the training accuracy percentage by calling `Boosting.evaluate()`.
4. Do the same for the testing data. Create predicted labels by selecting N random numbers $\in \{-1, 1\}$ where N is the number of testing images. Report its accuracy percentage.
5. Use the trained `WeakClassifier` to predict the testing data and report its accuracy.
6. Use the trained `Boosting` classifier to predict the testing data and report its accuracy.

Code:
- `ps6.Boosting.train()`
- `ps6.Boosting.predict(X)`
- `ps6.Boosting.evaluate()`

Report:
- Text Answer: Report the average accuracy over 5 iterations. In each iteration, load and split the dataset, instantiate a Boosting object and obtain its accuracy.
- Text Answer: Analyze your results. How do the Random, Weak Classifier, and Boosting perform? Is there any improvement when using Boosting? How do your results change when selecting different values for num_iterations? Does it matter the percentage of data you select for training and testing (explain your answers showing how each accuracy changes).

3. Haar-like Features [20 points]

In this section you will work with Haar-like features which are normally used in image classifications. These can act to encode ad-hoc domain knowledge that is difficult to learn using just pixel data. We will be using five types of features: *two-horizontal, two-vertical, three-horizontal, three-vertical, and four-rectangle*. You may want to review the contents in Lesson 8C-L2 and the Viola-Jones paper. In this section you will complete the `HaarFeature` class.

a. You will start by generating grayscale image arrays that contain these features. Create the arrays based on the parameters passed when instantiating a `HaarFeature` object. These parameters are:
- `type`: sets the type of feature among *two_horizontal, two_vertical, three_horizontal, three_vertical, and four_square* (see examples below).
- `position`: represents the top left corner **(row, col)** of the feature in the image.
- `size`: **(height, width)** of the area the feature will occupy.

Complete the function `HaarFeatures.preview()`. You will return an array that represents the Haar features, much like each of the five shown above. These Haar feature arrays should be based on the parameters used to instantiate each Haar feature. Notice that, for visualization purposes, the background must be black (0), the area of addition white (255), and the area of subtraction gray (126). Note that the area occupied by a feature should be evenly split into its component areas - three-horizontal should be split into 3 evenly sized areas, four-square should be split into 4 evenly sized areas (in other words, divide the width and height evenly using int division).

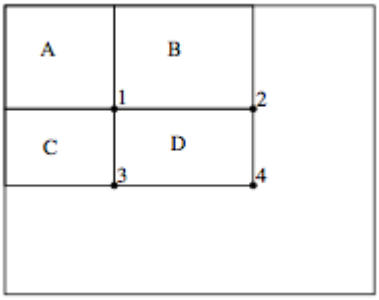
Code: `Functions in HaarFeatures.preview()`

Report: Using 200x200 arrays.
- Input: type = two_horizontal; position = (25, 30); size = (50, 100) Output: **ps6-3-a-1.png**
- Input: type = two_vertical; position = (10, 25); size = (50, 150) Output: **ps6-3-a-2.png**
- Input: type = three_horizontal; position = (50, 50); size = (100, 50) Output: **ps6-3-a-3.png**
- Input: type = three_vertical; position = (50, 125); size = (100, 50) Output: **ps6-3-a-4.png**
- Input: type = four_square; position = (50, 25); size = (100, 150) Output: **ps6-3-a-5.png**

b. As you may recall from the lectures, the features presented above represent areas that would either add or subtract the image area depending on the region "colors". The white area will represents an addition and the gray area a subtraction. In order to follow the class content, we will work with Integral images. The integral image at location x, y contains the sum of the pixels above and to the left of x, y, inclusive. Complete the function `ps6.convert_images_to_integral_images`.

Code: `ps6.convert_images_to_integral_images(images)`

c. Notice that the step above will help us find the score of a Haar feature when it is applied to a certain image. Remember we are interested in the sum of the pixels in each region. Using the procedure explained in the lectures you will compute the sum of the pixels within a rectangle by adding and subtracting rectangular regions. Use the example from the Viola-Jones paper:



Assume the image above is an array ii returned as an integral image. The sum of pixels in D can be obtained by $sum(D) = ii(4) - ii(2) - ii(3) + ii(1)$ where each number corresponds to the pixel position shown in the image. If D was a white rectangle in a Haar feature you will use it as $+D$, if it was a gray rectangle then it becomes $-D$.

Important note: Points 1, 2, and 3 in the example above, do not belong to the rectangle D. They cover the areas just before the one in D. Test your approach with the result obtained from D = sum(D) + sum(A) - sum(B) - sum(C).

Complete `HaarFeatures.evaluate(ii)` obtaining the scores of each available feature type. The base code maps the feature type strings to the following tuples (you will see this in the comments):
- "two_horizontal": (2, 1)
- "two_vertical": (1, 2)
- "three_horizontal": (3, 1)
- "three_vertical": (1, 3)
- "four_square": (2, 2)

Code: `HaarFeatures.evaluate(ii)`

Report:
- Text Answer: How does working with integral images help with computation time? Give some examples comparing this method and np.sum.

4. Viola-Jones [30 points]

In this part we will use the features obtained in the previous section and apply them in face recognition. Haar-like features can be used to train classifiers restricted to using a single feature. The results from this process can be then applied in the boosting algorithm explained in the Viola-Jones paper. We will use a dataset of images that contain faces (pos/ directory) and refer to them as positive examples. Additionally, use the dataset of images in the neg/ directory which contain images of other objects and refer to them as negative examples. Code the boosting algorithm in the `ViolaJones` class.

First, we will be using images resized to 24x24 pixels. This set of images are converted to integral images using the function you coded above. Instantiate a `ViolaJones` object using the positive images, negative images, and the integral images. `ViolaJones(train_pos, train_neg, integral_images)`. Notice that the class

contains one attribute to store Haar Features (`self.haarFeatures`) and another one for classifiers (`self.classifiers`).

We have provided a function `createHaarFeatures` that generates a large amount of features within a 24x24 window. This is why it is important to use integral images in this process.

Use the Boosting algorithm in the Viola-Jones paper (labeled Table 1) as a reference. You can find a summary of it below adapted to this problem set.

For each integral image, evaluate each available Haar feature and store these values.

Initialize the positive and negative weights $w_{pos} = \frac{1}{2p}$ $w_{neg} = \frac{1}{2n}$ where p and n are the number of negative and positive images respectively. Stack them into one list w_t

For $t = 1, \ldots, T$: T: number of classifiers (hypotheses)

1. Normalize the weights,
$$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
2. Instantiate a and train a classifier h_j using the `VJ_Classifier` class.
   For each feature, j, obtain the error evaluated with respect to w_t:
   $\epsilon_j = \sum_i w_i$ for cases where $h_j(x_i) \neq y_i$
   ϵ_j can be obtained from the error attribute in `VJ_Classifier`
   Train h_j using the class function `train()`. This will fix the classifier that returns lowest error ϵ_t.
3. Append h_j to the `self.classifiers` attribute.
4. Update the weights:
   $w_{t,i} \leftarrow w_{t,i}\beta_t^{1-e_i}$ $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$ $e_i = negative$ if $h_t(x_i) == y_i$, $e_i = positive$ otherwise
5. Calculate:
$$\alpha_t = log(\frac{1}{\beta_t})$$
   Append it to the `self.alphas`.

Using the best classifier h_t, a strong classifier is defined as:

$$H(x) =$$

- *Positive label* if $\sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2}\sum_{t=1}^{T} \alpha_t$
- *Negative label* otherwise

a. Complete `ViolaJones.train(num_classifiers)` with the algorithm shown above. We have provided an initial step that generates a 2D array with all the feature scores per image. Use this when you instantiate a `VJ_Classifier` along with the labels and weights. After calling the weak classifier's train function you can obtain its lowest error $\epsilon$by using the 'error' attribute.

Code: `ViolaJones.train(num_classifiers)`

b. Complete `ViolaJones.predict(images)` implementing the strong classifier $H(x)$ definition.
Code: ViolaJones.predict(images)

Report:
- Output: **ps6-4-b-1.png** and **ps6-4-b-2.png** which correspond to the first two Haar features selected during the training process.
- Text Answer: Report the classifier accuracy both the training and test sets with a number of classifiers set to 5. What do the selected Haar features mean? How do they contribute in identifying faces in an image?

c. Now that the ViolaJones class is complete, you will create a function that is able to identify faces in a given image. In case you haven't noticed we're using images that are intended to solve the problem below. The negative directory contains patches of a scene where there is people in it except for their target face(s). We are using this data to bias the classifier to find a face on a specific scene in order to reduce computation time (you will need a larger amount of positive and negative examples for a more robust classifier).

Use a 24x24 sliding window and check if it is identified as a face. If this is the case, draw a 24 x 24 rectangle to highlight positive match. You should be able to only find the man's face. To that extent you will need to define a positive and negative datasets for the face detection. You can choose images from the pos/ and neg/ datasets or build your own from the man.jpeg image by selecting subimages.

Code: `ViolaJones.faceDetection(image, filename=None)`

Report: Use the following input images and return a copy with the highlighted face regions.

- Input: **man.jpeg**. Output: **ps4-4-c-1.png**

5. Challenge Problem [N/A]

There is(are) no challenge problem(s) in this problem set.