CSC487/2503 – Computational Vision – Fall 2008

Assignment 3: Modeling and Detecting Human Eyes

Due: at the beginning of the tutorial, 12:10 pm., Mon., Nov. 10
This assignment is worth 15 marks towards your grade in this course.

This assignment explores principal component analysis (PCA) for representing images, along with two alternative approaches for view-based object detection.

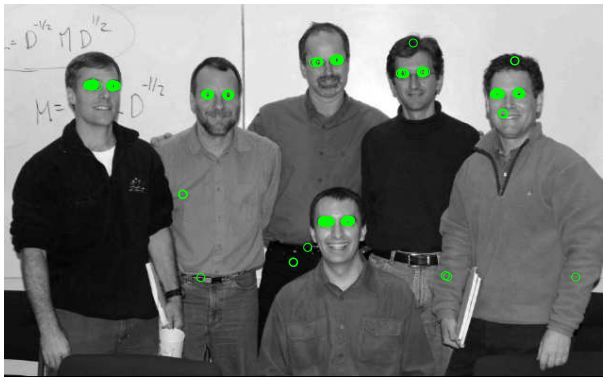

Figure 1: Results of an eye detector roughly similar to one of the detectors constructed in this assignment. Green circles indicate detected eyes. Your results will vary.

Download `A3Handout.zip` from the course web page.

What to hand in. Write a short report addressing each of the itemized questions below (hand-written reports are fine). You can assume that the marker knows the context of the questions, so do not spend time repeating material in the hand-out or in class notes. Include print outs of the output from your programs in your report. Also, please email `mstefan@cs.utoronto.ca` each of the Matlab files that you altered or created.

Training Data. One large set of eye and non-eye images has been split into two disjoint sets, one in `trainSet.mat`, the other in `testSet.mat`. We will use the former to train various eye detectors and, once the training is complete, we will use the latter to test their performance.

The images of eyes have been warped to be roughly of constant position, orientation, and scale (the scale was set by specifying an inter-eye distance of about 40 pixels). These warped images were then cropped to be of size 20×25. They are represented as 500 dimensional columns. There is also a set of non-eye images in the training and test sets, which are used to tune and test the detectors.

The handout code `trainEigenEyes.m` loads the training set of images. The mean brightness of each image is subtracted, and the result is scaled so that the variance is equal to the number of pixels, namely 500. Then the mean eye image is subtracted out. This results in a training set of normalized difference images, say $D_k(\vec{x})$ for $k = 1, \ldots, K$, for which the PCA components are computed. The SVD then produces the basis images $B_j(\vec{x})$ along with the singular values σ_j, for

$j = 1, \ldots, 500$. In order to match the principal covariances of the data set, the singular values have been rescaled by $\frac{1}{\sqrt{K}}$.

Save the results in `svdEyes.mat` (line 129 in `trainEigenEyes.m` does this, but it is not run by default).

1. **Presmoothing.** The script `trainEigenEyes.m` allows the image set to be filtered with a Gaussian filter before scaling and computing the SVD. We define the *effective dimension* of a data set as the smallest dimension containing 95% of the total variance. What are the effective dimensions of the eye data set when: a) no preblur is used; b) a preblur of $\sigma = 2/\pi$ is applied; and c) $\sigma = 4/\pi$ is used? (This connects back to our discussion of dimension on p.43 of the lecture notes on Linear Filters, Sampling, and Fourier Analysis.) Describe in words the effect of using $\sigma = 2/\pi$ or $4/\pi$ on the magnitude of the singular values. Are all singular values scaled by the same amount? (When you are finished this question, make sure that you save `svdEyes.mat` for the case in which no preblur is used. We won't use presmoothing in any of the questions below.)

2. **Residual Variance.** Suppose we consider approximating the training images $D_k(\vec{x})$ with just the first n PCA basis images, $B_j(\vec{x})$ for $j = 1, \ldots, n$. Define the residual variance:

$$V_n(\vec{x}) \equiv \frac{1}{K} \sum_{k=1}^{K} \left[D_k(\vec{x}) - \sum_{j=1}^{n} b_{k,j} B_j(\vec{x}) \right]^2, \tag{1}$$

   where the expansion coefficient $b_{k,j}$ is given by $b_{k,j} = \sum_{\vec{x}} D_k(\vec{x}), B_j(\vec{x})$. Show mathematically that this residual variance $V_n(\vec{x})$ satisfies

$$V_n(\vec{x}) = \sum_{j=n+1}^{500} \left[B_j(\vec{x}) \sigma_j \right]^2, \tag{2}$$

   where σ_j^2 is the j^{th} eigenvalue of the data covariance matrix associated with the eigenvector $B_j(\vec{x})$.

   At the end of `trainEigenEye.m` add some code to form $V_n(\vec{x})$, for $n = 20$ and $n = 50$ and display the results as images. Computationally check that (1) and (2) almost identical results (with the difference being plausibly due to rounding errors). Roughly where in the image is the residual variance the largest?

3. **EigenEye Detector.** The second script file, namely `trainDetector.m`, uses the above PCA results to develop an eye detector. In particular, let $I(\vec{x})$ be a rescaled test image with,

$$\sum_{\vec{x}} I(\vec{x}) = 0, \quad \sum_{\vec{x}} I^2(\vec{x}) = nx * ny,$$

   where $ny = 25$ and $nx = 20$ are the number of rows and columns in the image. In order to detect whether or not $I(\vec{x})$ is the image of an eye (at the right position, orientation, and scale) we first subtract out the mean eye image $M(\vec{x})$ (computed and saved by `trainEigenEyes.m`). The result is the difference image,

$$D(\vec{x}) = I(\vec{x}) - M(\vec{x}). \tag{3}$$

   We expand this using the PCA basis $\{B_j(\vec{x})\}_{j=1}^{n}$, giving the model image

$$A_n(\vec{x}) = \sum_{j=1}^{n} B_j(\vec{x}) a_j. \tag{4}$$

Here the coefficients are $a_j = \sum_{\vec{x}} B_j(\vec{x})D(\vec{x})$. Finally the error in this model is

$$E(\vec{x}) = D(\vec{x}) - A_n(\vec{x}). \tag{5}$$

On page 15 of the lecture notes on View-Based Models we describe a simple approach for detecting eyes based on the coefficients a_j, for $j = 1, \ldots, n$, and the error $E(\vec{x})$. Here we take a somewhat different approach. For each rescaled image $I(\vec{x})$ define the 4-vector

$$\vec{X} = (1, ||D(\vec{x})||^2, S_{in}, S_{out})^T \tag{6}$$

where

$$||D(\vec{x})||^2 \;=\; \sum_{\vec{x}} D^2(\vec{x}), \;\; S_{in} \;=\; \sum_{j=1}^{n} (a_j/\sigma_j)^2, \;\; S_{out} \;=\; \sum_{\vec{x}} (E(\vec{x})/V_n(\vec{x}))^2.$$

Note the first component of $\vec{X}$ is simply one, which is convenient for fitting a planar decision boundary to this data. Here n is the dimension of the eigenspace to be used. Modify `trainDetector.m` so that it generates this statistics vector $\vec{X}$ for each training image, both the eyes in `testTarg` and the non-eyes in `testNon`.

The next block of code in `trainDetector.m` generates a 3D scatter plot of the results, then does a logistic regression to fit a simple classifier to this data. The logistic regression fits a parameter vector $\vec{\beta}$ for the classifier

$$p(\vec{X}) = \frac{e^{\vec{\beta}^T \vec{X}}}{1 + e^{\vec{\beta}^T \vec{X}}}. \tag{7}$$

Here a value of $p(\vec{X}) > \tau$ indicates the image should be categorized as an eye, for some threshold τ.

Add code to `trainDetector.m` to generate an ROC plot for five separate detectors, each fit with logistic regression as described above, using eigenspace dimensions $n = 1, 5, 10, 20, 50$. Points along a single ROC curve are obtained by thresholding $p(\vec{X})$ in (7) at different values in the range $[0, 1]$. How do these detectors compare with the ones used for the ROC plots on p.17 of the View-Based Models lecture notes?

Finally, we need to evaluate these detectors on a separate test set, which is loaded from the file `testSet.mat`. The point of this test set is to evaluate the performance of your newly developed detectors on a data set that it has never seen before. This allows you to check that you have not overfit the detector to the training data. The idea is that the results on the test set should be similar to those you get on the training set. Add code to the end of `trainDetector.m` to compute the statistics vectors, $\vec{X}$, for each image in this test set, and evaluate $p(\vec{X})$. (Do NOT refit your detectors on this test set!) Do you get similar ROC plots to the ones for the training set?

Your report should include the ROC plots for both the training and test sets. How do the detection results compare for different dimensions n of the eigenbasis? Do you get a consistent improvement as the dimension n increases to 50? Try to suggest reasons why, or why not.

4. **Weak Classifiers.** In this question and the next we will demonstrate an eye classifier built using the AdaBoost learning algorithm. This is a boosted classifier, based on many weak classifiers. Your task in this question is to complete `trainStump.m`, which sets some of the parameters for a weak classifier.

   The input for this M-function the function is a projection vector $\vec{f}$, a flag `useAbs`, an $N \times K$ dimensional data matrix X, $1 \times K$ class label vector y (with $y(k)$ equal to 0 or 1, indicating

the k^{th} column of X corresponds to a non-target or target, respectively), and a $1 \times K$ vector of non-negative weights w. We seek a "weak-classifier" of the form:

$$h(\vec{x}, \vec{\theta}) \;=\; \begin{cases} I(u(\vec{f}^T\vec{x}) \leq \theta_1), & \text{for } \theta_2 = 1, \\ I(u(\vec{f}^T\vec{x}) > \theta_1), & \text{for } \theta_2 = -1, \end{cases} \tag{8}$$

where $u(z)$ is either just z or $|z|$, depending on whether or not the flag *useAbs* is false or true. Also, $I(b)$ in (8) is 1 when the boolean value b is true, and 0 otherwise. Note, this weak classifier $h(\vec{x}, \vec{\theta})$ is similar to the weak classifier $f(\vec{x}, \vec{\theta})$ on p.27 of the View-Based Models notes, except there the classifier takes values $+1$ and -1, while here it is $+1$ and 0.

The only unknowns for `trainStump.m` are the threshold value θ_1 and the parity θ_2. Complete the M-function `trainStump.m` so that it returns the values described in the function comment, including the optimum parameters $\vec{\theta}$. Denoting the k^{th} column of X by $\vec{x}_k$, the optimum $\vec{\theta}$ should minimize the weighted error

$$err \;=\; \frac{\sum_k w_k I(y_k \neq h(\vec{x}_k, \vec{\theta}))}{\sum_k w_k} \tag{9}$$

Hint: You can find θ_1 and θ_2 in Matlab without writing your own loop by using the built-in functions `cumsum`, `sum`, `max`, and/or `min`.

Complete a short Matlab script, `testStump.m`, to demonstrate your M-function `trainStump.m`. In particular, build several weak classifiers based on projection vectors $\vec{f}$ obtained from derivatives of a Gaussian kernel (i.e., generated by `buildGaussFeat.m` in the handout code). You can assume the weights in $\vec{w}$ are all equal (although it is important that your implementation of `trainStump.m` works correctly for any non-negative weights with $\sum_k w_k \neq 0$). In your report show histograms of the continuous-valued function $u(\vec{f}^T\vec{x})$ for both target $\vec{x}$'s and non-targets. Report the true positive rate and the false positive rate of each of your weak classifiers using the optimal $\vec{\theta}$.

5. **AdaBoost.** Browse the script file `trainAdaGauss.m`. The next thing you need to do to complete the training part of this script is to write `evalBoosted.m` as described in the handout code's comments. Given a list of weak classifiers $\{h_m(\vec{x}, \vec{\theta}_m)\}_{m=1}^M$, which have been trained as in question 5 above and found to have errors err_m, for $m = 1, \ldots, M$, then the strong classifier is given by

$$S_M(\vec{x}) = \sum_{m=1}^{M} \alpha_m (2h_m(\vec{x}, \vec{\theta}_m) - 1), \tag{10}$$

here $\alpha_m = \log((1 - err_m)/err_m)$. (This expression is different from the one on p.27 of the lecture notes, since our weak classifiers have discrete values in $\{0, 1\}$ instead of $\{-1, 1\}$.)

The first part of the script file `trainAdaGauss.m` can then be run to train a boosted classifier for eyes. This iteratively adds one weak classifier to the additive linear model (10), trying many different weak classifiers to determine which one to add. This search over many weak classifiers may take a minute or so per classifier so you might try using only `nFeatures = 20` weak classifiers. If you have more CPU resources, try training `nFeatures = 50` or `100`.

As each weak classifier is selected, it is particularly instructive to observe the histograms of the feature before thresholding, that is, histograms of $u(\vec{f}^T\vec{x})$, for both the targets and the non targets. (The M-file `trainStump.m` in the handout code is set up to make these plots.) If there is an edge in most of the training target images at a particular scale, sign, and orientation, what might you expect a selected feature to be? If the sign of the edge varies (i.e., either light to dark or dark to light), what might you expect a selected feature to be? If the training target images

are all relatively smooth (i.e., no edges) in a particular region, what might you expect a selected feature to be?

Add code to the end of `trainAdaGauss.m` to compute the ROC curves obtained by thresholding the strong classifier in (10), that is, $S_M(\vec{x}) > \tau$, for different values of τ. Draw separate ROC curves for various values of M. Do this for both the training set of eyes and non-eyes, and the test set. To evaluate $S_M(\vec{x})$ for some M less than the maximum number of features you trained, you do not need to rerun the training. Why? Simply limit the sum in (10) to the first M terms. Comment on the results. Are the testing and training errors for a given strong classifier (i.e., the same M) similar? If not, explain what the cause for this might be. How do the performances of the AdaBoost trained detectors compare to your previous eigen-eye detectors?

6. **Application to an Image Patch.** The script file `tryEyeDetector.m` reads in an image of several people, none of whom appear in the previous training or test sets. Write Matlab code to select a rectangle from this image (using ginput(2)) and run an eye detector centered at every pixel within the selected rectangle. You can use either an eigen-eye detector such as the one developed in question 3, or an AdaBoost eye detector such as the one in question 5 above. Show the resulting detections overlayed on top of the original image.

   The reason for not running the eye detector over the whole image is that these Matlab implementations are slow. You can write a relatively efficient Matlab implementation by storing the image as one long column vector I, and then forming $X = I(J)$, where J is a $500 \times K$ matrix of indices into I. If you choose the indices in J correctly, each column of X will be the pixels from a 20×25 patch within I. The detector can then be applied to all the columns in X quite efficiently. This trick won't work for the whole image, since you will run out of memory, but it will be able to process many patches at a time (i.e., K patches).

   Small Print. The eye detection results in Figure 1 were obtained using a similar positive set to the one used here, but with a much larger negative set. The main point of showing these results here is to demonstrate that a plausibly useful result can be obtained with the techniques described in this assignment.

7. **Synthetic Visual Cortex.** Suppose the first stage of our computational vision system always computed an image pyramid. And say this pyramid included Gaussian filtered images (say at σ equal to 4, 2, and 1), along with the first and second Gaussian derivatives (i.e., g_x, g_y, g_{xx}, g_{xy} and $g_y y$) at each of these scales. Completely specify the operations required to compute any of the weak classifiers (8) actually used in Question 6 above. (I want a written answer, you do not need to program anything.) Here you can assume you are given $\vec{n} = (\cos(\theta), \sin(\theta)$ corresponding to the steering orientation, θ, of any directional derivative filter required. Note, some of the filters used in `buildGaussFeat.m` were truncated to fit within the 20×25 image patch. You can ignore this spatial truncation in describing how the weak classifiers are computed from these pyramids. Also, roughly how many arithmetic operations are required to compute the strong classifier $S_M(\vec{x})$ above (ignoring the costs of forming and indexing into the image pyramids, and the effects of spatial filter truncation)?